

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER : H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees,
H. Niemann, D. Seitzer, H.J. Schneider

Entwurf und Bewertung von Algorithmen für ein
mikroprogrammiertes Parallel- und Assoziativ-
konzept auf Mikroprogramm- und Benutzerebene

Heide Wichmann

Dissertation

BAND 10 · NUMMER 16 · ERLANGEN · DEZEMBER 1977

1978 # 28

ARBEITSBERICHTE
DES INSTITUTS FÜR MATHEMATISCHE
MASCHINEN UND CYBERNETIK
(INFORMATIK)

VERGLEICHENDE STUDIEN ÜBER DIE
MATHEMATISCHE THEORIE DER
ALGEBRA UND DER LOGIK

Digitized by the Internet Archive
in 2026

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

**HERAUSGEBER : H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees,
H. Niemann, D. Seitzer, H.J. Schneider**

**Entwurf und Bewertung von Algorithmen für ein
mikroprogrammiertes Parallel- und Assoziativ-
konzept auf Mikroprogramm- und Benutzerebene**

**Dem Fachbereich Ingenieurwissenschaften
an der Friedrich-Alexander-Universität Erlangen-Nürnberg
zur Erlangung des Grades
Doktor - Ingenieur**

**vorgelegt
von
Heide Wichmann**

Dissertation

BAND 10 · NUMMER 16 · ERLANGEN · DEZEMBER 1977

ARBEITSBERICHT
DES INSTITUTS FÜR MATHEMATISCHE
MASCHINEN UND DATENVERARBEITUNG
(INFORMATIK)

VERGLEICHENDE STUDIEN ÜBER DIE GRUNDLAGEN DER
INFORMATIONSTHEORIE
VON: Dr. H. J. Schneider, Dr. W. Händler, Dr. D. Seitzer
Betreiber: Dr. H. J. Schneider, Dr. W. Händler, Dr. D. Seitzer
Lehrstuhl: Dr. H. J. Schneider, Dr. W. Händler, Dr. D. Seitzer

Entwurf und Bewertung von Algorithmen für ein
mikroprogrammiertes Parallel- und Asynchron-
konzept auf Mikroprogramm- und Rechnerebene



Als Dissertation genehmigt vom
Fachbereich Ingenieurwissenschaften
der Universität Erlangen-Nürnberg

Tag der Einreichung: 22. September 1977

Tag der Promotion: 7. Dezember 1977

Dekan: Prof. Dr. D. Seitzer

Berichterstatter: Prof. Dr. W. Händler

Prof. Dr. H.J. Schneider

Zusammenfassung

Parallelarbeit, im Sinne vieler gleichartiger Rechenwerke, und assoziativer, inhalt-gesteuerter Datenzugriff erscheinen als diejenigen Systemeigenschaften, um die der konventionelle Universalrechner erweitert werden sollte, um eine große Klasse von Spezialaufgaben angemessen bearbeiten zu können, und die sich mit vertretbarem Aufwand durch Mikroprogramme realisieren lassen. Neben der wortorientierten Emulation wird die Vertikalverarbeitung betrachtet; die Ergebnisse beider Ansätze werden einander gegenübergestellt und mit entsprechenden Werten für unkonventionelle hardware-realisierte Systeme verglichen. Anschließend werden Algorithmen auf Benutzer-Niveau untersucht und zwar speziell Algorithmen für Graphen, bei denen sich die Kombination der Systemeigenschaften - assoziativer Zugriff und parallele Verarbeitung - als nützlich erweist.

An dieser Stelle möchte ich all jenen danken, die dazu beigetragen haben, daß ich diese Arbeit nun vorlegen kann. Herrn Professor Händler danke ich für das Interesse, das er meiner Arbeit jederzeit entgegengebracht hat. Zu Dank verpflichtet fühle ich mich gegenüber Herrn Professor Herzog und Herrn Professor Schneider. Meinen Kollegen danke ich für viele anregende Diskussionen. Und ein besonderes Dankeschön gilt Frau Pastore für ihre gewissenhafte Schreibaarbeit.

Inhaltsverzeichnis

	Seite
1. <i>Einleitung</i>	1
1.1 Problemstellung	1
1.2 Überblick über die Arbeit	2
2. <i>Konzept der Vertikalverarbeitung</i>	4
2.1 Kurze Beschreibung der Vertikalverarbeitung. Anwendungsbeispiele	4
2.2 Abgrenzung gegen verwandte Konzepte: STARAN, OMEN und DAP	6
2.3 Voraussetzungen für die Implementierung der Vertikalverarbeitung	7
2.3.1 Relevante Eigenschaften der Standard-Hardware	8
2.3.2 Einige Überlegungen zur Software	11
2.3.3 Notwendige Zusatz-Hardware zur Datenkonversion	13
2.3.3.1 "Drehscheibe"	14
2.3.3.1 Orthogonalspeicher	16
3. <i>Entwurf einer Befehlsliste für das mikroprogrammierte Parallel- und Assoziativkonzept und ihre effektive Implementierung</i>	17
3.1 Vergleichbare unkonventionelle Spezial-Hardware	17
3.1.1 Assoziative Systeme	18
3.1.1.1 Kurze allgemeine Beschreibung	18
3.1.1.2 Beispiel einer Addition	20
3.1.2 Arithmetisches Pipeline-Array	24
3.2 Allgemeine Überlegungen zu den mikroimplementierten Algorithmen	29
3.3 Notation	31
3.3.1 Darstellung von Zahlen und Zeichen	31
3.3.2 Beschreibungssprache	33

3.4	Befehlslisten für einzelne Anwendungsklassen. Implementierungsskizzen	35
3.4.1	Vektoren-Arithmetik	37
3.4.1.1	Addition/Subtraktion	41
3.4.1.2	Zweierkomplement	44
3.4.1.3	Multiplikation	45
3.4.1.4	Division	51
3.4.1.5	Quadratwurzel	52
3.4.1.6	Komponentensumme	54
3.4.1.7	Komponentenweiser Vergleich	56
3.4.1.8	Arithmetisches Maximum/Minimum	58
3.4.1.9	Schwellwertvergleich	62
3.4.1.A	Zählen addierend/subtrahierend	62
3.4.2	Assoziative Verarbeitung	63
3.4.2.1	"Exact Match"	63
3.4.2.2	"Best Match"	66
3.4.2.3	Einschreiben	68
3.4.2.4	Lexikalisches Maximum/Minimum	69
3.4.2.5	Schwellwertvergleich	69
3.4.3	Musterverarbeitung	69
3.4.4	Hilfsoperationen	70
3.5	Auswertung	71
3.5.1	Laufzeitvergleiche für verschiedene Maschinenkonzepte und Implementierungen	72
3.5.2	Forderungen nach Zusatz-Hardware zur Unterstützung der Vertikalverarbeitung	82
3.5.2.1	"Vertikal-Modul"	82
3.5.2.2	"Quersummen-Modul"	86
3.5.2.3	Zusatz-Modul zur Verarbeitung binärer Muster	90
3.5.2.4	Spezieller Speicherprozessor	91
4.	<i>Entwicklung von Graphenalgorithmen für Parallel- und Assoziativrechner. Vergleich für verschiedene Rechnerkonzepte</i>	99
4.1	Kurze Zusammenfassung von einigen numerischen. Anwendungsbeispielen	99
4.2	Algorithmen für Graphen	101
4.2.1	Vergleichbare unkonventionelle Spezial-Hardware: zelluläre Arrays	103
4.2.2	Notation zur Darstellung der Algorithmen für das Parallel- und Assoziativkonzept	107

	Seite
4.2.3	Aufspannender Baum eines Graphen 110
4.2.3.1	Test auf starken Zusammenhang 110
4.2.3.2	Maximaler (minimaler) aufspannender Baum 113
4.2.3.3	Aufspannender Baum und seine Fundamentalzyklen 115
4.2.4	Kürzeste Wege in einem Graphen 117
4.2.4.1	Alle Wege in einem Graphen 119
4.2.4.2	Alle kürzesten Wege in einem Graphen 120
4.2.5	Eliminationsgraphen 123
4.2.5.1	Auffüllen eines Graphen 124
4.2.5.2	Perfekte Ordnung eines perfekten Eliminationsgraphen 128
4.2.5.3	Minimale Eliminationsordnung eines Graphen 131
4.2.6	Hamilton-Wege in einem Graphen 138
4.2.6.1	Alle Hamilton-Wege in einem Graphen 139
4.2.6.2	Kürzester Hamilton-Weg in einem Graphen 140
4.3	Auswertung der Algorithmen für Graphen 143
4.3.1	Zusammenfassung und Interpretation der Vergleiche 147
4.3.2	Bedeutung des mikroprogrammierten Parallel- und Assoziativkonzeptes für die Graphenalgorithmen 149
<i>Anhang A: Konventionelle Emulation des Parallel- und Assoziativkonzeptes</i>	151
A.1	Addition 151
A.2	Multiplikation 152
A.3	Maximumsuche 154
A.4	"Best Match" 156
<i>Anhang B: Numerische Anwendungsbeispiele für das Parallel- und Assoziativkonzept: Potenzenberechnung</i>	158
<i>Literatur</i>	160
<i>Liste der Identifikatoren in den Mikroprogramm-Algorithmen</i>	166
<i>Register</i>	168
<i>Verzeichnis der Abbildungen</i>	iv
<i>Verzeichnis der Tabellen</i>	v

Verzeichnis der Abbildungen

Nr.		Seite
2.1	Konzept der Vertikalverarbeitung	5
2.2	Horizontale und vertikale Darstellung eines Datenblockes und Konversion mit "Drehscheibe"	15
3.1	Schema eines assoziativen Speichers	18
3.2	Datenbereich und Steuerinformation mit "Immediate-Feld" nach der Initialisierung (nach /56/)	22
3.3	Programm zur Vektoren-Addition in einem funktionalen Speicher (nach /56/)	23
3.4	Schema der Subtrahenden/Addenden bei Quadratwurzel/Quadrat (nach /27/)	25
3.5	a) Addierer-/Subtrahierer-Zelle b) Steuer-Zelle (nach /27/)	26
3.6	Schema eines verallgemeinerten Pipeline-Arrays (nach /27/)	27
3.7	Beispiel einer vertikalen Vektoren-Addition/Subtraktion	40
3.8	"Vertikal-Modul" neben Registersatz und Standard-ALU in einem Rechner mit Modul-Konzept vom Typ der Interdat 85	83
3.9	"Quersummen-Modul" zwischen den Bussen in einem Rechner mit Modul-Konzept vom Typ der Interdata 85	88
3.A	Adreß- und Datenregister zwischen Zentral- und Speicherprozessor	94
3.B	Puffer zum Lesen und Schreiben zwischen Zentral- und Speicherprozessor	97
4.1	Zellulares Array zum Test auf starken Zusammenhang (nach /31/)	105
4.2	Zellulares Array zur Bestimmung eines aufspannenden Baumes (nach /31/)	106

Verzeichnis der Tabellen

Nr.		Seite
3.1	Vergleich der horizontalen Blockoperationen und der Vertikaloperationen	73
3.2	Vergleich der gesamten Laufzeiten und der expliziten Speicherzugriffszeiten bei der Vektoren-Addition - verschiedene Implementierungen und unterschiedliche Speicherzugriffszeiten	93
3.3	Vergleich der gesamten Laufzeiten und der Zeitverhältnisse von Berechnungszeit zu Speicherzugriffszeit bei der Vektoren-Addition - verschiedene Implementierungen in einem System mit zusätzlichem Speicherprozessor	96
4.1	Vergleich der Algorithmen für verschiedene Maschinenkonzepte nach Laufzeitverhalten und Speicheraufwand für die Daten	145

1. Einleitung

1.1 Problemstellung

Dem klassischen Universalrechner mit gemeinsamem linear adressierten Speicher für Programm und Daten, einer arithmetisch-logischen Verknüpfungseinheit und ein paar Registern stehen heute eine Reihe unkonventioneller Rechner gegenüber, die wir im wesentlichen beschreiben können als Rechner mit mehreren verschiedenen oder gleichartigen parallel arbeitenden Verknüpfungswerken und Rechner mit assoziativem Datenspeicher ohne oder mit Verarbeitungseinheiten zu jedem einzelnen Speicherwort. Außerdem gibt es Mehrrechnersysteme, in denen gleiche oder verschiedene Rechner verkoppelt sind. Die unkonventionellen Rechner und Rechnersysteme sind konzipiert teils ebenfalls für universellen Einsatz, teils für Spezialaufgaben. Gerade diese Systeme sind dann nur sehr schwerfällig zu gebrauchen für andere als ihre Spezialaufgaben.

Der Versuch liegt nahe, Universalität zu kombinieren mit unkonventionellen Systemeigenschaften und als Werkzeug dazu die *Mikroprogrammierung* zu benutzen. Zu erhoffen sind höhere Leistung bei Spezialaufgaben und besser solchen Aufgaben angepaßte Programmierung, zusätzlich eine höhere Flexibilität, etwa in Datenumfang und Format, verglichen mit hardware-realisierten Spezialrechnern. Unter Umständen kann die Mikroprogrammierung unterstützt werden durch Zusatz-Hardware, so wie im klassischen Universalrechner viele Operationen, die früher mikroprogrammiert wurden, heute auch bei Minirechnern hardware-implementiert sind.

Parallelarbeit, im Sinne vieler gleichartiger Rechnerwerke, und assoziativer Zugriff erscheinen als diejenigen Systemeigenschaften, die für viele Spezialaufgaben nützlich sind und die sich mit vertretbarem Aufwand auf einem konventionellen Rechner durch Mikroprogramme realisieren lassen. Versucht man, für die möglichen Anwendungsgebiete abgerundete Befehlslisten zu entwerfen, und analysiert die einzelnen Operationen, so zeigt sich, daß die meisten Operationen "linear" sind, das soll heißen: daß die Bits der Operanden höchstens einmal verarbeitet werden, und daß bei allen parallel verknüpfenden Operationen eine dateninterne Sequentialisierung möglich ist, das soll heißen: daß Einzeldaten oder Datenpaare bit-seriell verarbeitet werden können

ohne Kenntnis der gleichzeitig verarbeiteten anderen Daten. Deshalb lohnt es sich, hier als Alternative zur üblichen Emulation unkonventioneller Operationen die *Vertikalverarbeitung* zu untersuchen, wie sie W. Händler /15/ 1974 vorgeschlagen hat.

Im Gegensatz zum gewohnten Entwurfsverfahren, wo man, ausgehend von den vorgesehenen Anwendungen, über die Menge der Algorithmen für ihre Bearbeitung, zu einem Maschinenkonzept kommt und weiter, über die Menge der realisierenden Mikroprogramm-Algorithmen, zu dessen Hardware-Grundlage - wobei Hardware und Maschinenkonzept ihrerseits die möglichen Algorithmen bedingen -, haben wir hier den Gedankengang umgekehrt: wir beginnen mit der konventionellen Standard-Hardware, implementieren durch Mikroprogramme ein Parallel- und Assoziativ-Konzept und untersuchen dann die Anwendungsmöglichkeiten. Übrigens haben auch wir eine starke Rückkopplung zwischen zugrundeliegenden Maschinenkonzepten und möglichen Algorithmen berücksichtigt. Deshalb wurde versucht, sowohl für die Mikroprogramm-Algorithmen wie für die Algorithmen auf Anwenderebene Bewertungen zu geben.

1.2 Überblick über die Arbeit

In Kapitel 2 wird das Konzept der Vertikalverarbeitung beschrieben. Außerdem ist dort zusammengetragen, was im Laufe der Pilot-Implementierung als wesentliche Voraussetzungen für die praktische Arbeit erkannt wurde. Die Pilot-Implementierung ist bereits früher /21/ dokumentiert worden, Kurz skizziert werden vergleichbare Konzepte.

Kapitel 3 beschäftigt sich intensiv mit Mikroprogramm-Algorithmen. Um Vergleichsdaten für die Bewertungen zu haben, werden zunächst kurz ein assoziatives System und eine arithmetische Pipeline skizziert. Danach werden Befehlslisten für mögliche Anwendungsgebiete zusammengestellt. Anschließend werden Algorithmen für die Vertikalimplementierung der Befehle vorgestellt: Es kam darauf an, Algorithmen zu finden, die es erlauben, einen gesamten Datenbestand homogen zu bearbeiten, zum Beispiel ohne Fallunterscheidungen nach Vorzeichen. Es gelingt, eine Reihe von sehr "kompakten" Algorithmen zu finden: Algorithmen, die es erlauben, mehrere Vertikaloperationen mit ein und demselben Mikroprogramm auszuführen, wobei durch binäre Variable

ausgewählt wird, sogar dynamisch, also simultan für verschiedene Operanden bzw. Operandenpaare verschiedene Operationen auszuführen. Einige Algorithmen der vergleichbaren konventionellen Emulation stehen im Anhang. Als Hilfsmittel zur Notation der Algorithmen wird eine höhere Programmiersprache benutzt, die es erlaubt, "räumliche" und "zeitliche" Wiederholungen parametrisiert darzustellen sowie asynchrone Überlappungen. Sehr bald zeigt sich, daß zwar die Struktur der Algorithmen "elegant" ist, daß aber bei der Ausführung die konventionelle Hardware zu schwerfällig ist. Deshalb werden spezifizierte Forderungen nach Zusatz-Hardware gestellt. Die Algorithmen mit ihren verschiedenen Hardware-Voraussetzungen werden benutzt zu Laufzeitabschätzungen, die ihrerseits eine Rechtfertigung der Zusatz-Hardware liefern. Zusatz-Hardware sind hier Komponenten vom Typ einer arithmetisch-logischen Einheit, die neben der Standard-ALU eingebaut werden und alternativ zu ihr angesteuert werden können, aber auch Hilfsmittel, um die vielen Speicherzugriffe effektiv abzuwickeln.

Anschließend (in Kapitel 4) wird mit Hilfe von Algorithmen auf Anwenderebene untersucht, ob die beiden Mikroprogramm-Implementierungen des Parallel- und Assoziativ-Konzeptes gleichwertig sind und wo die qualitativen Unterschiede zu hardware-realisierten assoziativen Systemen liegen. Bei numerischen Anwendungen spricht alles zugunsten der konventionellen Emulation, vor allem, daß dort Gleitkomma-Arithmetik möglich ist. Bei den Graphen-Algorithmen gelang es, einige Algorithmen nachzubilden, die ursprünglich für zellulare Arrays geschrieben waren, und andere analog zu formulieren, auch solche, die sich auf Arrays nicht formulieren ließen. Hier ist die Vertikalverarbeitung zu favorisieren, einmal wegen der einfacheren Arbeit mit Masken, zum anderen, weil ihre Hilfs-Hardware, vor allem der Orthogonalspeicher, sich als ausgezeichnete Spezial-Hardware für den Bit-Matrizen-Kalkül erweist. Bei den Laufzeitabschätzungen sieht man den wesentlichen Unterschied gegenüber einem echten assoziativen System, der vor allem in der Reduktion der Parallelität auf höchstens r (Länge eines Speicherwortes bzw. Rechenwerkbreite) besteht. Diese Einschränkung ist so stark, daß gelegentlich sequentielle Algorithmen auf einem klassischen Maschinenkonzept effektiver implementiert werden können als parallele Algorithmen auf dem mikroprogrammierten Parallel- und Assoziativkonzept. Es zeigt sich aber auch, daß es Probleme gibt, zu deren Lösung assoziative Hilfsmittel, ganz gleich wie sie realisiert sind, nichts Wesentliches beitragen. Das mikroprogrammierte Konzept hat hier seinen besonderen Wert als Experimentiersystem, also Auswahl- und Vorbereitungshilfe für unkonventionelle Hardware.

2. Konzept der Vertikalverarbeitung. Voraussetzungen für ihre Implementierung

2.1 Kurze Beschreibung der Vertikalverarbeitung. Anwendungsbeispiele

Betrachten wir den Hauptspeicher eines konventionellen Computersystems mit w Worten der Breite r als $w \times r$ -Bitmatrix. Die arithmetisch-logische Einheit verarbeitet Daten aus dieser Bitmatrix zeilenweise. Der konventionellen - "horizontalen" - Verarbeitung liegt auch eine zeilen-orientierte Interpretation zugrunde. So sind etwa die Bits einer Matrixzeile die Binärstellen einer Festkommazahl im Zweierkomplement. 1974 schlug W. Händler /15/, zurückgreifend auf Ideen aus den Sechzigerjahren /14/, die Vertikalverarbeitung vor: $\ell \times r$ -Teilmatrizen ($\ell \leq w$) im Speicher sollen interpretiert werden als Block von r gleichformatigen Daten der frei wählbaren Länge ℓ . Das heißt: Teile von Spalten der Bitmatrix werden als zusammengehörig interpretiert. Auch diese Daten werden selbstverständlich zeilenweise verarbeitet. Benutzt man dabei aber nur logische Operationen, verhält sich das r -Bit-Parallelrechenwerk wie r identische isoliert, aber parallel arbeitende 1-Bit-Rechenwerke. Wir können damit - bezogen auf die Daten, das heißt: die Interpretation - wort-parallele bit-serielle Verarbeitung implementieren. (Abbildung 2.1)

Im wesentlichen lassen sich zwei Klassen von Operationen implementieren: Parallel-Arithmetik zur Verknüpfung von Blöcken (Vektoren) von Operanden, wobei je Operandenpaare aus derselben Matrixspalte verknüpft werden zu einem Ergebnis, das in dieser Spalte abgelegt wird; - und assoziative Operationen. Beide Operationenklassen arbeiten mit Auswahlmasken, die durch ein Bit pro Spalte markieren, ob der Operand oder das Operandenpaar in dieser Matrixspalte in der aktuellen Operation bearbeitet werden soll. Insbesondere dürfen Eintragungen im Ergebnisbereich nicht verändert werden, wenn eine Spalte nicht ausgewählt ist.

Ob diese Parallel- und Assoziativoperationen die konventionelle Befehlsliste ersetzen oder erweitern sollen, hängt sehr stark von den vorgesehenen Anwendungen ab. Prinzipiell erlaubt die Vertikalverarbeitung, weil sie ausschließlich mikropimplementiert ist, eine flexible Konfiguration aus konventionellen und unkonventionellen Systemkomponenten. Mögliche Anwendungen können zum Beispiel sein: Vektorenkalkül im Sinne der Mathematik, Datenbankbearbeitung, Unterstützung der Compiler, Lösung von Problemen der Betriebsmittelverwaltung.

Hauptspeicher:
w Worte, r Bits

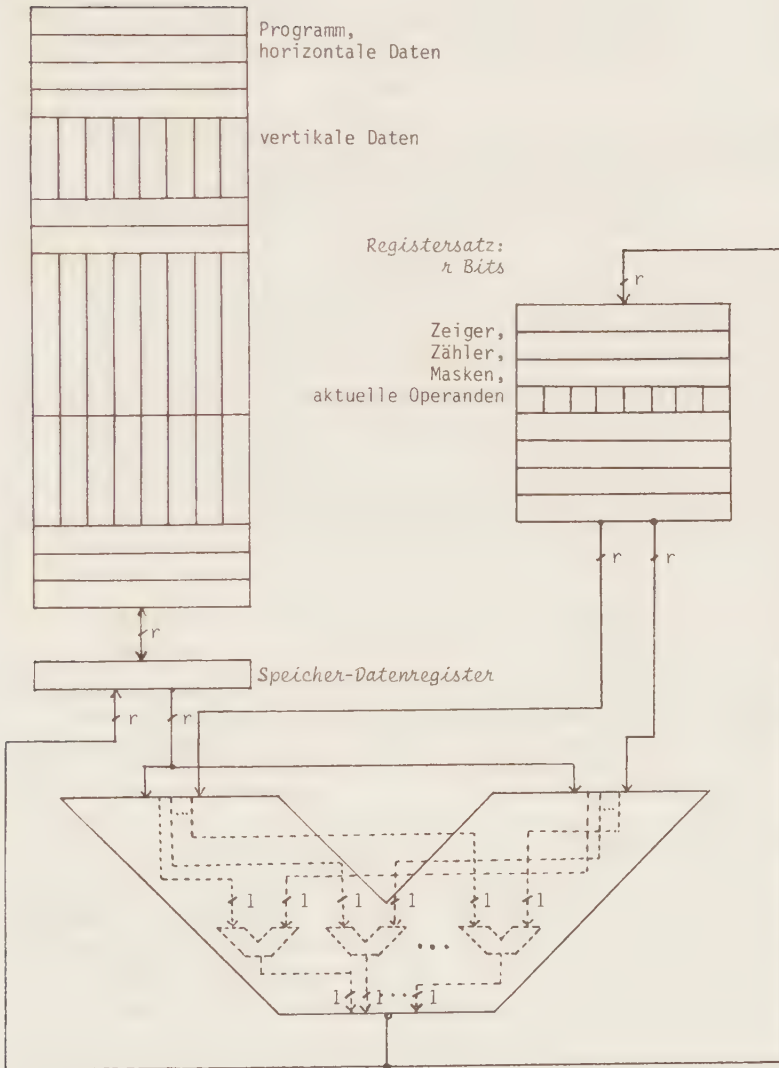


Abbildung 2.1: Konzept der Vertikalverarbeitung

Bei der Verknüpfung vertikal gespeicherter Datenblöcke arbeitet das r-Bit-Parallelrechenwerk wie r identische synchrone 1-Bit-Seriellrechenwerke.

Der kritische Punkt des Konzeptes ist die Datenkonversion, das heißt: das Einspeichern der vertikalen Daten sowie das Auslesen etwa der vertikalen Ergebnisse. Die ausschließlich mikroprogrammierte Lösung ist sehr zeitaufwendig (vgl. 2.2.3).

Gelegentlich mag es *spezielle Anwendungen* geben, wo die Daten bereits vertikal eingegeben werden können. Denken wir etwa an ein Steuerungssystem mit r parallelen Meßfühlern, wo die Daten im Takt digitalisiert bit-seriell an den Computer übergeben werden, beginnend mit dem höchstwertigen Bit. Dann können bereits bei der Datenübernahme "assoziative" Such- und Vergleichsoperationen ausgeführt werden, etwa Vergleiche mit einem Schwellwert oder Extremumsuche. Das Ergebnis wäre eine Markierung derjenigen Meßfühler, für deren Meßwert eine bestimmte Bedingung gilt; es könnte in ein Steuersignal umgeformt und wieder ausgegeben werden. Soll nicht weiter mit diesen Daten gerechnet werden, brauchen sie nicht einmal gespeichert werden. Für arithmetische Verknüpfungen wie Addition und Subtraktion müssen die Daten mit dem niedrigstwertigen Bit voraus ankommen, wenn sie in Echtzeit verarbeitet werden sollen; für nichtlineare Operationen wie die Multiplikation müssen die Daten zwischengespeichert werden.

Analog diesem Anwendungsbeispiel kann man Spezialperipherie konzipieren als Massenspeicher für ein System, in dem ausschließlich vertikal gearbeitet wird, zum Beispiel eine Platte mit r parallel arbeitenden Leseköpfen. Auch hier können viele der Vertikaloperationen bei geeignet gewählter Abspeicherungsrichtung "real-time" während der Datenübertragung ausgeführt werden.

2.2 Abgrenzung gegen verwandte Konzepte: STARAN, OMEN und DAP

STARAN /52/ hat einen in Quadrate zerlegten Orthogonalspeicher und ein lineares 1-Bit-Prozessoren-Feld (mit 1-Bit-Hilfsregistern). Das Permutations-Netzwerk, das zur zweifachen (orthogonalen) Adressierung des Speichers notwendig ist, steht für explizite Permutationsoperationen, sogenannte "Spiegelungen", zur Verfügung. Die große Speicherbreite ($r=256$) und die komplexe Hardware, die viele Vorgänge, die in konventionellen Rechnern mikroprogrammiert ablaufen, automatisch ausführt, machen STARAN zu einem kräftigen Spezialrechner, der im allgemeinen in ein Mehrrechnersystem eingebettet ist, zumindest an einen Universalrechner gekoppelt wird.

OMEN /51/, das ich hier nach dem ursprünglichen Konzept beschreibe, hat ebenfalls einen Orthogonalspeicher, der aber in 32·1024-Bit-Teilmatrizen eingeteilt ist. Neben dem konventionellen 32-Bit-Parallelrechenwerk gibt es ein 1024-Bit-"Vertikalverknüpfungswerk". Dieses Verknüpfungswerk enthält allgemeine Hilfs- und spezielle Übertragungsregister. Es kann alle 16 möglichen logischen Zwei-Operanden-Verknüpfungen ausführen und außerdem, als Drei-Operanden-Befehl, einen Additionsschritt mit Einschluß des Übertrages. Die Datenübertragung zwischen Speicher und diesem Verknüpfungswerk ist parallel für je eine 1024-Bit-Teilspalte und zwar maskiert.

Auch DAP ("Distributed Array Processor") /11/, /55/ verarbeitet Daten "vertikal". Während wir aber beim konventionellen Rechner eine Bit-Matrix der Dimensionalität 2 als Speicher haben, gibt es bei DAP einen Speicherkubus, eine Säule mit quadratischer Grundfläche. Unserem linearen Feld von (scheinbar isolierten) 1-Bit-Prozessoren steht dort ein quadratisches Feld von 1-Bit-Prozessoren (mit 1-Bit-Hilfsregistern) gegenüber. Die quadratisch angeordneten Prozessoren mit den zugehörigen linearen Speicherbereichen können in zwei Richtungen mit Daten versorgt werden, was zum Beispiel sehr effektiven Matrizenkalkül erlaubt. DAP als ganzes ist ein Spezialprozessor und soll in einer konventionellen Konfiguration etwa einen Speichermodul ersetzen.

2.3 Voraussetzungen zur Implementierung der Vertikalverarbeitung

Mit dem Anspruch der ursprünglichen Idee, auf konventioneller Hardware nur durch neue Mikroprogramme unkonventionelle Systemeigenschaften zu erreichen, stoßen wir rasch an praktische Grenzen: Nicht jede Hardware ist gleich gut für unsere Zwecke zu verwenden. Und es gibt wichtige Funktionen - wie etwa die Datenkonversion -, die sich mit konventionellen Hilfsmitteln nur sehr mühsam realisieren lassen. Deshalb soll sich dieser Abschnitt mit wünschenswerten Hardware-Voraussetzungen beschäftigen, wozu speziell auch Mikroprogrammier-Voraussetzungen gehören sollen.

Dieselben Überlegungen - bis auf die Zusatz-Hardware - gelten auch für jedes andere komplexe mikroprogrammierte System mit langen Laufzeiten zwischen den Decodierphasen, also etwa auch für die konventionelle Emulation des Parallel- und Assoziativkonzeptes.

2.3.1 Relevante Eigenschaften der Standard-Hardware

W. Reimer /46/ und R. Eck /8/ haben in ihren Diplomarbeiten einige der Vertikaloperationen, die in der Pilot-Implementierung auf der Interdata 85 konkret zur Verfügung stehen, auf andere mikroprogrammierbare Rechner übertragen, vor allem um diejenigen Eigenschaften der Rechner zu finden, die für die Vertikalverarbeitung wichtig sind. Aus beiden Arbeiten stehen eine Reihe von quantitativen Ergebnissen zur Verfügung, vor allem vergleichbare Meßergebnisse für Laufzeiten, Speicherzugriffe usw. Die Interpretation dieser Ergebnisse soll hier zusammengefaßt werden.

1. Der Rechner muß *mikroprogrammierbar* sein. Wahrscheinlich genügt es nicht, einmal entwickelte und ausgetestete Mikroprogramme für die Vertikalverarbeitung in einem ROM zur Verfügung zu stellen, weil ein abgerundeter Befehlssatz voraussichtlich sehr viele und vergleichsweise lange Mikroprogramme erfordern wird, die unter Umständen gar nicht alle gleichzeitig (und zusammen mit dem Mikroprogrammen für den Standard-Befehlssatz) in den Mikroprogrammspeicher passen. Im allgemeinen sind solche Speicher nur sehr beschränkt ausbaufähig, haben zum Beispiel nicht genügend Adreßraum. Deshalb müssen wir davon ausgehen, daß es nötig sein wird, die Mikroprogramme nach Bedarf nachzuladen, etwa in Modulen für verschiedene Anwendungen.

2. Die Mikroprogramme zur Vertikalverarbeitung sind wesentlich komplexer als etwa die Standard-Mikroprogramme einer Maschine vom IBM/360-Typ. Insbesondere benötigt jede Vertikaloperation eine ganze Reihe von Parametern zur exakten Beschreibung der Operanden (Adressen, Längen, Masken) und der Operation selbst (z.B. Codierungsmasken), die während des gesamten Ablaufs des Mikroprogrammes immer wieder gebraucht werden. Zusätzlich braucht man Zähler, Rückkehradressen bei Sprüngen usw. Alle diese Informationen, aber auch aktuelle Operandenwerte und Zwischenergebnisse sollten maximal schnell verfügbar sein. Das bedeutet: Um für die Vertikalverarbeitung gut geeignet zu sein, sollte der Rechner *möglichst viele* (für den Mikroprogrammierer) *frei benutzbare Register* haben, eventuell zusätzlich noch hardwaremäßig spezialisierte Register für Sonderaufgaben. Ein System mit hardware-implementierten Push-down-stack statt Registerfile ist bisher nicht in Betracht gezogen worden, erscheint aber wenig geeignet, die umfangreiche Buchführung über die Daten im Speicher effektiv abzuwickeln.

3. In den quantitativen Ergebnissen hat sich als wesentlicher Vorteil gezeigt, wenn es sich bei den Datenpfaden zum und vom Rechenwerk um ein *Drei- statt Zwei-Bus-System* handelt, wenn also Drei- statt Zwei-Adreß-Mikrobefehle möglich sind, aber auch, wenn etwa alle Register dabei gleichermaßen adressierbar sind.

4. Ein zentrales Kriterium ist, ob es sich im weitesten Sinne um *horizontal oder vertikal codierte Mikrooperationen* handelt. Einerseits bietet die horizontale Mikroprogrammierung die Möglichkeit, optimal alle Hardware-Parallelismen des Rechners auszunutzen und etwa durch unkonventionelle Kombinationen von Mikrooperationen gegenüber jedem Ablauf von Standard-Benutzerbefehlen die gewünschten Funktionen schneller leisten zu können. Andererseits würde bei den modernen Rechenanlagen ein horizontales Steuerwort sehr viele Bit erfordern. Das würde insbesondere die Entwicklung umfangreicher Mikroprogrammpakete sehr kompliziert machen. Die rein vertikale Mikroprogrammierung ist aber auch nicht sehr interessant, weil die Hardware sozusagen "eingefroren" ist auf einen Teil der überhaupt möglichen Funktionen, vor allem aber, weil in sehr hohem Maß sequentialisiert werden muß, was hardware-mäßig parallel gearbeitet werden könnte. Die meisten Rechner der Leistungsklasse, die uns hier interessiert, haben eine Mikrocodierung, die zwischen "horizontal" und "vertikal" einzuordnen ist: Während etwa die Rechenwerkfunktionen standardisiert sind, können Rechenwerk- und Speicherfunktionen nebeneinander angestoßen werden; eventuell gibt es auch mehr als diese zwei Moduln im Zentralsystem, die per Mikroprogramm mehr oder weniger unabhängig nebeneinander gesteuert werden können. Die Rolle eines solchen Moduls spielt zum Beispiel die Erzeugung der Folgeadresse im Mikroprogramm: Entweder das automatische Adressen-Hochzählen, wie es inzwischen in vielen Rechnern vorkommt, mit gelegentlichen bedingten oder unbedingten Sprüngen, wobei die Sprungbedingung und das Sprungziel zum Beispiel in jedem Mikrobefehl enthalten sind oder aber als spezieller Mikrobefehl in der Sequenz vorkommen, - oder die explizite Angabe einer Folgeadresse in jedem Mikrobefehl oder sogar zweier alternativer Folgeadressen und ihrer Verzweigungsbedingung.

5. Ebenso wichtig ist der *Vorrat an Mikrobefehlen*: Wir benutzen intensiv logische Operationen, und es ist wichtig, möglichst viele davon einfach zur Verfügung zu haben, statt sie durch mehrere andere aufbauen zu müssen. Im weiteren Sinne zum Vorrat an Mikrobefehlen zähle ich auch das Spektrum an möglichen Sprungbedingungen: Auch hier ist es angenehm, möglichst viele Bedingungen unmittelbar auswerten zu können. W. Reimer /46/ hat festgestellt, daß es auf jeden Fall wichtig ist, den Wert eines Bits in einem Datenwort

als Sprungbedingung verwenden zu können. In vielen Fällen kann man sich dabei durch Abfrage des Übertrag-Bit nach einem Schiebebefehl behelfen. Wesentlich ist auch, daß Konstanten als Operanden benutzt werden können.

6. Man kann wohl davon ausgehen, daß alle uns interessierenden Rechner erlauben, Speicheroperationen von Mikrobefehlen anzustoßen, die auch noch andere Funktionen steuern. Aber die Konzepte unterscheiden sich darin ganz erheblich, ob *Speicher und Steuerwerk synchron oder asynchron* arbeiten. Ganz besonders im Fall der synchronen Zusammenarbeit kommt es dann auch auf die Geschwindigkeitsverhältnisse an, wieviele ALU-Operationen während eines Speicherzugriffes ausgeführt werden können usw. Günstig erscheinen Konzepte, bei denen Speicheroperationen und Rechenwerkoperationen einander so aufhalten, daß mit wohldefinierten Daten gearbeitet werden kann. Außerdem sind diejenigen Konzepte zu favorisieren, bei denen die Speichergeschwindigkeit etwa durch Austausch gegen einen Speicher in einer neuen Technologie erhöht werden kann. Das Spektrum der Speichersteuerfunktionen, zum Beispiel automatische Adreß-Hochzählung usw., ist ebenfalls interessant.

7. Um dynamisch austauschbare Mikroprogramm-Moduln laden zu können, haben wir einen *beschreibbaren Steuerspeicher* gefordert. Es wäre angenehm, auch Daten aus dem aktuell laufenden Mikroprogramm dort ablegen zu können, zum Beispiel alte Inhalte der Benutzer-Register oder Parameterwerte, die selten gebraucht werden. Leider bietet sich diese Möglichkeit nur bei ganz wenigen Rechnern. Bei den meisten wird der Mikroprogrammspeicher wie ein externes Speichermedium geladen; zum "Einlesen" steht nur der Datenpfad über das Mikrobefehlsregister zur Verfügung, das heißt, es können nur die Mikrobefehle in der Reihenfolge ihrer Abarbeitung gelesen werden, aber keine Daten. Damit scheidet für solche Rechner auch ein echter Mikroprogrammaustausch aus, bei dem nicht einfach der alte Steuerspeicher-Inhalt überschrieben wird, sondern ausgetauscht.

8. Für ein System, das nicht nur zum Entwickeln und Experimentieren im Bereich der Vertikalverarbeitung dienen soll, sondern für reale Anwendungen sind *außer den relativen Geschwindigkeiten auch die absoluten* interessant.

9. Sollen die Vertikaloperationen in einem realen System verwendet werden unter einem "beliebigen" realen Betriebssystem, müssen sie unterbrechbar werden, weil ihre Laufzeiten viel größer sein können, zum Beispiel bei langen

Daten, als die Bedien-Zeiten mancher Ein-/Ausgabegeräte. Das bedeutet: es muß auf der Mikroprogrammebene ein klares *Unterbrechungs-Konzept* geben, das erlaubt, lange Mikroprogramme zu unterbrechen mit einem Minimum an Verwaltungsaufwand zur Zustandsbeschreibung und zum exakten Wiederaufsetzen. Wünschenswert ist ein Konzept, bei dem man abfragen kann auf anstehende Unterbrechungssignale, also Meldung in ein Spezialregister ohne sofortige Unterbrechung; man kann die Unterbrechung aufschieben zum Beispiel bis zu einer Stelle im Mikroprogramm, wo relativ wenig Parameter und Zwischenwerte gerettet zu werden brauchen.

10. Zum Schluß ein Kriterium, das für die beiden Diplomarbeiten /46/, /8/ nicht relevant erschien, aber nach unseren Erfahrungen mit der Pilot-Implementierung auf der Interdata 85 nicht zu unterschätzen ist: Zwar bleibt der Anspruch des Konzeptes, mit konventioneller Hardware auszukommen; aber zum Beispiel während der Experimentierphase interessiert, ob es *Hardware-Alternativen oder Erweiterungen* gibt, die die Vertikalverarbeitung besonders begünstigen. Deshalb kann es eine sehr wichtige Frage sein, ob die Standard-Hardware zu vertretbarem Aufwand Änderungen oder Erweiterungen erlaubt. Das Modul-Konzept der Interdata 85 zum Beispiel, bei dem "konkurrierend" zur Standard-ALU weitere "Moduln" eingebaut und angesteuert werden können, ist hier eine sehr praktische Lösung.

2.3.2 Einige Überlegungen zur Software

Hier sollen nur kurz einige Aspekte skizziert werden, die sich nach den bisherigen Erfahrungen und Überlegungen als wichtig gezeigt haben.

1. Um ein komplexes Mikroprogrammpaket entwickeln zu können, braucht man einen Benutzer-freundlichen *Mikroassembler* und ein *Testsystem*, das erlaubt, den Mikrozustand der Maschine zu beobachten, möglichst auch: gezielt im Detail zu verändern. Die Interdata 85, auf der die Pilot-Implementierung erstellt wurde, bietet den besonderen Vorteil, daß nicht nur das Hardware-Konzept die Erweiterung der Zentraleinheit um Zusatz-Moduln erlaubt, die konsistent mit der Standard-Hardware angesteuert werden können, sondern auch der Mikroassembler dafür eingerichtet ist, neue Mikrobefehle zu übersetzen.

2. Wir brauchen einen *Mikroprogramm-Lader*, möglichst sogar einen Betriebssystem-Modul, der einen effektiven Mikroprogramm-Austausch erlaubt. Falls es irgend der Mikrocode erlaubt, sollten die Mikroprogramme verschieblich sein, um die Teile eines größeren Mikroprogrammpaketes möglichst frei kombinieren zu können. Diese Voraussetzung ist allerdings wohl nur beschränkt zu erfüllen, weil zum Beispiel dort, wo Folgeadressen unmittelbar in einem Teil oder allen Mikrobefehlen codiert sind, diese "Folgeadressen" meistens nur Relativadressen etwa in einer "Seite" sind. In einem solchen Fall genügt aber bereits die Restriktion, daß Mikroprogramm-Moduln immer mit dem Anfang einer Seite beginnen.

3. Sollen die Mikroprogramme für die Vertikalverarbeitung in jedem beliebigen Auftragsstrom laufen, müssen sie wegen ihrer Länge unterbrechbar sein. Das bedeutet, daß das gesamte *Unterbrechungssystem* Rücksicht darauf nimmt, daß mehr Zustandskomponenten gerettet und rekonstruiert werden müssen als sonst, wo Unterbrechungen nur zwischen je zwei Mikroprogrammen (das soll heißen: während der Decodierphase) auftreten können. Zur Zustandsbeschreibung eines unterbrochenen Mikroprogrammes gehören viele Zeiger, das heißt: absolute Speicheradressen. Wenn man nicht nach jeder Unterbrechung den sehr aufwendigen Vorspann mit Parameterübernahme und -auswertung wiederholen will bzw. entsprechende Information über den Auftrag oder Abschnitt aus dessen Steuer-Block holen will (wobei man im Mikroprogramm statt über absolute Speicheradressen über Distanzwerte Buch zu führen hätte), muß man auf Austausch-Mechanismen im Betriebssystem verzichten. Das kann eine erhebliche Einschränkung bedeuten. Unabhängig davon müssen wir dafür sorgen, daß sichergestellt ist, daß unterbrochene Mikroprogramme innerhalb desjenigen Auftrags oder Abschnitts fortgesetzt werden, in dem sie unterbrochen wurden, das heißt sie müssen imstande sein, das aufrufende Programm zu identifizieren. Ein Betriebssystem mit Mikroprogramm-Austausch kann diese Identifizierung sozusagen garantieren. Man kann aber auch ein bestimmtes Mikroprogrammpaket wie ein einmalig vorhandenes Betriebsmittel genau einem Auftrag oder Abschnitt zuteilen und alle anderen, die es auch beantragen, solange warten lassen, bis es wieder freigegeben wird. Die Benutzung der "Drehzscheibe" (vgl. Abschnitt 2.3.3.1) weckte neue Schwierigkeiten: Hier handelt es sich um ein "unbekanntes" Ein-/Ausgabe-Gerät, dessen Ansteuerung niemandem außer dem Mikroprogrammierer erlaubt sein sollte. Das Betreiben über den Selektor-Kanal wäre zwar der Hardware-Geschwindigkeit der Drehzscheibe angemessen, würde aber dessen

Verwaltung durch das Betriebssystem unterlaufen und sollte deshalb unterbleiben. Bemerkung: Die Unterbrechungs-Verwaltung im Mikroprogramm bedeutet einen erheblichen Zusatzaufwand. Es sollte überlegt werden, ob man wirklich unter jedem beliebigen Betriebssystem in jeder beliebigen Programmumgebung die Vertikalverarbeitung zulassen will, oder ob man sie nicht wie die Anforderung eines aufwendigen Betriebsmittels, längerer Rechenzeit oder ähnliches behandeln will, das heißt: aufschieben bis zu einem geeigneten Zeitpunkt. Das hängt aber sehr davon ab, als wie universell brauchbar sich die Vertikalverarbeitung erweist (z.B. als Hilfsmittel für das Betriebssystem selbst, für die Compiler usw.).

4. Um die Vertikalverarbeitung dem Benutzer-Programmierer anbieten zu können, muß sie in einer *höheren Programmiersprache* verfügbar sein. B. Jahn hat in seiner Diplomarbeit /25/ eine Algol 68-Erweiterung vorgeschlagen. Es muß sich zeigen, wieweit die Mikroprogrammierung ihrerseits nützlich ist für die Implementierung von Compilern.

2.3.3 Notwendige Zusatz-Hardware zur Datenkonversion

Selbst wenn man große Sequenzen von Vertikaloperationen auszuführen hat, bei denen die Daten in immer derselben Darstellung verarbeitet werden, kommen bei einem konventionellen Computer Daten in konventioneller Form an, und sie sollen auch konventionell wieder aus- oder weitergegeben werden. Und sogar bei Anwendungen mit bit-sequentieller wort-paralleler Datenübernahme von irgendwelchen Meßleitungen usw. müssen die Ergebnisse für Protokolle usw. "lesbar" ausgegeben werden.

Das heißt: Wir brauchen einen effektiven Konversionsalgorithmus. Ein programmierter Algorithmus benötigt mindestens $O(n \cdot \ell)$ Operationen, wenn wir n Datenworte à ℓ Bits umsetzen wollen. Im folgenden sind zwei mögliche Hardware-Zusätze kurz skizziert, mit deren Hilfe man Datenblöcke umsetzen kann bzw. sich die Umsetzung sparen.

2.3.3.1 "Drehscheibe"

In der Pilot-Implementierung werden die notwendigen Konversionen mit der "Drehscheibe" gemacht. Das ist ein $r \times r$ -Bit-Externspeicher (hier: $r = 16$), in den r (konventionell betrachtete) Speicherworte ausgegeben werden und aus dem sie, abhängig vom Ein-/Ausgabe-Steuerswort, unter Vertauschung von Wort- und Bit-Adressierung und eventuell zusätzlicher Komplementierung von einer oder beiden Adressen wieder eingelesen werden. Damit sind Spiegelungen der r Speicherwörter, als Bit-Matrizen betrachtet, an beiden Diagonalen möglich sowie Drehungen um 90° in beiden Richtungen. Die "Drehscheibe" wurde nach Entwürfen von H. Hessenauer /19/ im Hardware-Seminar gebaut.

Die Umsetzung braucht $O(\frac{\ell}{r} \cdot n)$ Zeiteinheiten. Für die praktische Anwendung, etwa für ℓ Bit lange Daten, wo $\ell > r$ ist, werden die r Worte, die gemeinsam gespiegelt werden, ursprünglich nicht fortlaufend hintereinander gestanden haben, sondern sie sind jedesmal das k -te Wort von r horizontalen Eintragungen $\hat{a}_q = \left\lceil \frac{\ell \cdot n}{r} \right\rceil$ Worten, $q \geq k$, gewesen (vgl. Abbildung 2.2). Wir benötigen also zusätzlichen Aufwand für Buchführung. *)

Auch mit der Zusatz-Hardware ist die Konversion eines $r \times r$ -Blockes eine aufwendige Operation, da hier typische Ein-/Ausgabezeiten eingehen. Deshalb wird man versuchen, mit möglichst wenigen Konversionen auszukommen (etwa nur bei Eingabe der Daten und Ausgabe der Ergebnisse). Operationen "unter der Zeit", die Daten beider Darstellungen verknüpfen sollen, müssen dann mit anderen Hilfsmitteln realisiert werden. Wir haben dafür in der Pilot-Implementierung eine Reihe von Hilfsfunktionen wie Einfügen eines horizontal vorgegebenen Wortes in eine oder mehrere Spalten; Herauslesen einer Spalte zur Weiterverwendung in horizontaler Form usw. Solche Operationen lassen sich konsistent mikroprogrammieren. Eine Operation, für die das nicht mehr gilt, ist das "Packen": Nach Suchvorgängen etwa sind pro r Dateneintragen nur noch wenige interessant

Um den Grad an Parallelität bei der Weiterverarbeitung wieder zu erhöhen, werden die noch interessierenden Daten aus zwei solchen r -Blöcken zusammengepackt. Mit ausschließlich Mikroprogrammen als Implementierungsmittel ist diese Operation sehr aufwendig. In Kapitel 3 ist eine mögliche Zusatz-Hardware zur Unterstützung dieser Operation beschrieben.

*) $\lceil x \rceil$ ist die kleinste ganze Zahl, die größer als x ist

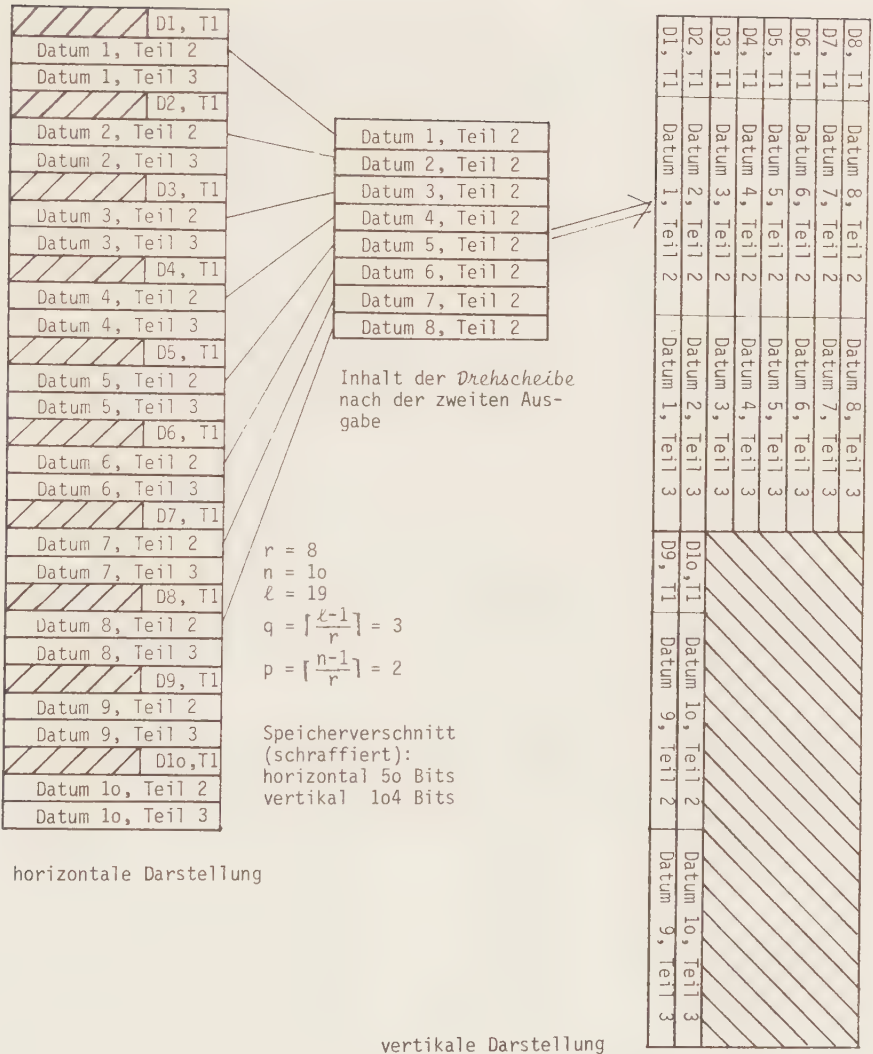


Abbildung 2.2: Horizontale und vertikale Darstellung eines Datenblockes und Konversion mit "Drehscheibe"

Es werden die jeweils entsprechenden Teilworte aus r Datenworten auf die Drehscheibe ausgegeben und unter Spiegelung an der Hauptdiagonalen wieder eingelesen.

in 2.3.2, Punkt 3, sind die prinzipiellen Probleme bei Benutzung der Drehscheibe skizziert: Es handelt sich um ein Ein-/Ausgabegerät, dessen Verwaltung dem Betriebssystem entzogen ist und bei dessen Ansteuerung über den Selektor-Kanal das Mikroprogramm dem Betriebssystem zum Beispiel die Status-Information über die zuletzt abgelaufenen Selektorkanal-Funktionen unter schlägt.

2.3.3.2 Orthogonalspeicher

Alternativ zur "Drehscheibe" hat H. Hessenauer /19/ einen "Orthogonalspeicher" vorgeschlagen, etwa als schnellen Arbeitsspeicherabschnitt für die Daten bei Vertikalverarbeitung.

Gehen wir von einem Speicher mit Wortlänge r aus, etwa $r=32$. Dann sei der Speicher in "Quadrate" von $r \times r$ Bits eingeteilt. Eine Speicheradresse gibt an, in welchem Quadrat ein Wort liegt, ob es "horizontal" oder "vertikal" liegt und welche Zeilen- oder Spaltennummer es hat. Bei einem solchen Konzept ist eine Datenkonversion prinzipiell nicht mehr nötig, aber vielleicht nahe- liegend, weil man dann den restlichen, konventionellen Speicher auch für vertikale Daten benutzen kann.

Einschreiben und Herauslesen von vertikalen Daten z.B. in Register sind triviale Operationen, und auch ein Packen ist ein einfaches Umspeichern. Dafür stehen die Drehoperationen für ein $r \times r$ -Quadrat nicht mehr hardware- mäßig zur Verfügung, sind aber einfach aufzubauen.

Bei manchen Anwendungen - z.B. bei Matrizen-Algorithmen für Graphen, vgl. Kapitel 4 - kann man für die Rechnungen die zweifache Zugriffsmöglichkeit unmittelbar ausnutzen.

Prinzipiell ist für die verknüpfenden Vertikaloperationen, z.B. die Vektoren- addition, ein lesender Zugriff auf die ursprünglichen Daten im Ergebnisbereich notwendig, um sicherzustellen, daß die ursprünglichen Daten in den nichtaus- gewählten Komponenten unbeschädigt erhalten bleiben (durch "maskiertes Mischen" mit den neuen Daten). Diesen zusätzlichen Speicherzugriff kann man sich sparen, wenn das Schreiben in den Speicher unmittelbar maskiert werden kann. Diese Möglichkeit hat H. Hessenauer in seinem Vorschlag /19/ vorgesehen.

3. Entwurf einer Befehlsliste für das mikroprogrammierte Parallel- und Assoziativkonzept und ihre effektive Implementierung

In diesem Kapitel soll das mikroprogrammierte Parallel- und Assoziativkonzept konkret durch seine Befehlsliste implementiert werden. Zwei Implementierungsansätze werden einander gegenübergestellt, die für den Benutzer nicht unterscheidbar, dasselbe leisten sollen: Vertikalverarbeitung und konventionell emulierte horizontale Blockoperationen (vgl. Anhang A). Aus den Implementierungsskizzen werden Laufzeitvergleiche abgeleitet. Zu diesen Vergleichen sollen außerdem Spezial-Hardware~~n~~ herangezogen werden, um mikroprogrammierte und hardware-realisierte Konzepte einander gegenüberstellen zu können. Als Folgerung aus den Laufzeitvergleichen wird sich die Forderung erheben nach Zusatz-Hardware zur Unterstützung der Vertikalverarbeitung.

3.1 Vergleichbare unkonventionelle Spezial-Hardware

In Abschnitt 2.2 werden bereits kurz drei der Vertikalverarbeitung ähnliche Hardware-Konzepte beschrieben. Für STARAN, OMEN und DAP kann man prinzipiell dieselben Algorithmen benutzen, wie sie später in diesem Kapitel für die Vertikalverarbeitung skizziert werden. Allerdings besteht dort, wo die 1-Bit-Prozessoren unabhängig voneinander angesteuert werden, nicht die Notwendigkeit, "homogene" Algorithmen zu benutzen, das heißt: Algorithmen, die unempfindlich sind gegen signifikante Eigenschaften der Daten wie etwa deren Vorzeichen. Wir können aber davon ausgehen, daß wir für solche Hardware-Konzepte qualitativ dieselben Laufzeitabschätzungen bekommen wie für die Vertikalverarbeitung.

Im folgenden sollen zwei andere Hardware-Konzepte kurz beschrieben werden, die ihrerseits "homogene" Algorithmen benutzen: assoziative Systeme und arithmetische Pipeline-Arrays. In beiden Konzepten sollen viele gleichartige Daten simultan oder zeitlich gestaffelt verarbeitet werden.

3.1.1 Assoziative Systeme

3.1.1.1 Kurze allgemeine Beschreibung

Das Grundkonzept des assoziativen Speichers besteht aus einem wortorientierten Speicherblock, einem Ein-/Ausgabe-Register, einem Masken-Register (beide so breit wie ein Speicherwort), pro Speicherwort einem Selektorbit; einer Treffer-Anzeige (1 Bit).

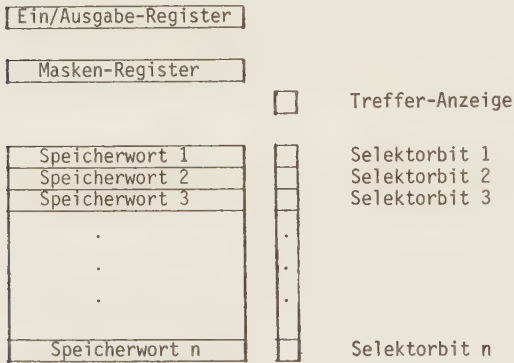


Abbildung 3.1: Schema eines assoziativen Speichers

Im Ein-/Ausgabe-Register stehen Daten, die eingetragen, gelesen, gesucht werden sollen. Im Masken-Register steht für diejenigen Bits eine 1, wo in den Speicherworten geschrieben, gelesen, gesucht werden soll - die anderen Bits werden als irrelevant betrachtet und weder gelesen noch verändert. Es werden alle Worte bearbeitet, die durch ihr Selektorbit ausgewählt sind.

Bei einem Suchvorgang ("Exact Match") etwa werden in allen ausgewählten Speicherworten die maskierten Bit-Positionen mit dem Suchkriterium im Ein-/Ausgabe-Register verglichen; für alle Speicherworte, wo Übereinstimmung festgestellt wurde, bleiben die Selektorbits auf dem Wert 1, die anderen werden gelöscht. Die Treffer-Anzeige läßt dann erkennen, ob noch mindestens ein Selektorbit gesetzt ist. Bei entsprechender Hardware dauert der eigentliche Suchvorgang (nach dem Aufsetzen der Register) "eine Zeiteinheit". Bei einem

System, das es erlaubt, mehrere aufeinanderfolgende Speicherworte zu verketten, zum Beispiel durch Verschieben des Selektors auf Nachbarwörter zuzugreifen, braucht man für ℓ Bit lange Daten $O(\frac{\ell}{r})$ Zeiteinheiten. Neben "Exact Match" wird für manche Anwendungen "Best Match" gebraucht und zwar nach verschiedenen Approximationskriterien. Eine mögliche "beste" Annäherung ist die mit der kleinsten Hamming-Distanz.

Die Hardware-Realisierung des "Exact Match" arbeitet mit einer Mimik zu jedem Speicherwort, die in jeder relevanten Stelle bei Nichtübereinstimmen eine 1 erzeugt, pro Wort alle "Mismatch"-Informationen durch Oder verknüpft, das Ergebnis negiert und mit dem alten Selektorbit durch Und verknüpft. Es gibt inzwischen Bausteine, die nicht das Oder der "Mismatch"-Informationen liefern, digital, sondern als Analog-Information die "Summe" dieser Informationen. Man könnte sich vorstellen, daß diese "Summen", überall, wo ein Selektorbit den Wert 1 hat, nacheinander verglichen werden mit, ebenfalls analogen, Werten 0 (das wäre "Exact Match"), 1, 2 usw., bis zum ersten Mal eine Übereinstimmung auftritt. Dort soll dann das Selektorbit gesetzt bleiben, während es überall sonst gelöscht wird. Der Schwellwert soll ablesbar sein (digital).

Dieses Konzept erfordert eine nichttriviale Hardware und kann sicher kein beliebig genaues Ergebnis liefern, da es, je mehr Werte möglich sind (prinzipiell von 0 bis r), desto schwieriger wird, sie zu trennen. Ob eine solche Hardware bereits realisiert ist, ist mir nicht bekannt.

Die Laufzeit eines solchen Systems muß sich mit $O(r)$ abschätzen lassen. Eine "Best Match"-Suche über Datenworte, die länger sind als r Bits, erscheint nicht mehr sinnvoll.

Für eine sinnvolle Nutzung eines solchen Speichers braucht man noch eine ganze Reihe von Aufsetzbefehlen (für Ein-/Ausgabe-Register, Masken-Register, Selektoren); man braucht Konventionen, was bei mehrfachen Treffern geschehen soll, etwa beim nächsten Lesen usw.

Programme und Mikroprogramme sind konventionell gespeichert und werden wie üblich sequentiell abgearbeitet.

Ein solcher assoziativer Speicher mit geeigneter Steuereinheit ist im allgemeinen eine Spezialkomponente in einem größeren System mit ansonsten konventionellen Komponenten. Je nach Anwendung und Fähigkeiten der Zentraleinheit nutzen wir dann die Möglichkeiten des schnellen (einfachen) Suchens in vielen Daten unmittelbar aus, etwa für ein Datenbanksystem, oder mittelbar zur Implementierung von komplexeren Such- oder Vergleichsoperationen oder auch arithmetischen Verknüpfungen. Dazu brauchen wir allerdings im allgemeinen zusätzliche Grundoperationen wie Shifts für die Register oder auch die ganze Selektor-Leiste; weitere Verknüpfungen der Treffer-Information aus den Worten in den Selektorbits (bisher: Und; jetzt zum Beispiel zusätzlich: Oder) und ähnliches. Eine solche Spezial-Komponente, die unter anderem für Parallel-Arithmetik eingesetzt ist, wird in 3.1.1.2 beschrieben.

Ist jedem Speicherwort hardwaremäßig ein Verarbeitungselement zugeordnet, sprechen wir von einem "assoziativen Prozessor". Wir können auf allen ausgewählten Speicherworten parallel dieselbe Operation ausführen und arbeiten dabei im einfachsten Fall bit-seriell, mit Auswahl des aktuellen Bits durch das Masken-Register.

In /7/ wird ein emulierter assoziativer Speicher (eigentlich: Prozessor) vorgestellt, der durch einen Spezialbefehl passend dimensioniert werden kann. In gewissem Sinn entspricht diese Emulation unserer Implementierung von horizontalen Blockoperationen. Die zugrundeliegende Hardware-Maschine hat aber für Hauptspeicher und Mikro-ROM ein Zeitverhältnis 10:1; deshalb sind die Ergebnisse dort - "assoziative" Programme verglichen mit konventionellen Programmen - mehr ein Argument für die Mikroprogrammierung als für das assoziative Konzept.

3.1.1.2 Beispiel einer Addition

In /56/ wird ein spezielles assoziatives System beschrieben, mit dem man Vektoren-Arithmetik implementieren kann. Gegenüber dem Grundkonzept, wie es in 3.1.1.1 kurz beschrieben ist, haben wir: Zwei Ein-/Ausgabe-Register (R1, R2); zwei Selektoren-Leisten (P, S); ein Maskenregister (M); zusätzlich als "virtuelle" Masken-Register: lauter Nullen ($\emptyset$), lauter Einsen (1); einen "virtuellen" Selektor mit lauter Einsen(1). Wir haben einen erweiterten Satz von Grundoperationen.

Der wesentliche Unterschied gegenüber dem einfachen assoziativen Speicher besteht darin, daß wir hier mit drei-wertiger Logik arbeiten: mit der Interpretation für die drei Werte "1", "0", "don't care". Entsprechend sind auch die Grundoperationen definiert. So wird etwa bei einem Suchvorgang ein Bit im Speicher, das den Wert "don't care" hat, sowohl bei Vergleich mit "1" wie auch bei Vergleich mit "0" als "passend" bewertet (also als Treffer). *)

Ein 4-Bit-"Immediate-Feld" der (Mikro-)Operationen wird mit den obersten vier Bits des angesprochenen Ein-/Ausgabe-Registers und des (realen oder "virtuellen") Masken-Registers geodert und zwar genau für die laufende Operation, ohne Änderung der Register selbst. Dadurch hat man unter anderem ein Hilfsmittel, "Daten" und "Steuerinformation" im Speicher zu markieren und zu erkennen, auch weitere Markierungen zu setzen. Die "Steuerinformation" wird zur Steuerung des (Mikro-)Programmablaufs als Satz von Masken benutzt.

Die Daten: Für das Beispiel, das hier vorgeführt werden soll, werden (beliebig viele) Paare von je gleichlangen Operanden je in ein Speicherwort geschrieben mit einer Markierung (im obersten der vier Spezialbits) als "Daten". Es wird im Zweierkomplement gearbeitet. Das zweitoberste Markierungsbit ist mit 0 initialisiert und markiert später "aktiv"/"inaktiv"; das dritte, ebenfalls mit 0 initialisiert, markiert später "Übertrag". Das vierte Bit wird hier nicht benutzt.

Bei den Masken brauchen wir außer dem obersten Bit ("Daten"/"Steuerinformation") noch die Markierung: "Beginn des Maskenfeldes" (im 4. Bit des Immediate-Feld).

Im Maskenbereich stehen am Anfang zwei Masken zur Trennung der Operanden; dann eine Folge von 1-Bit-Masken zum Auswählen je eines Bits in einem der beiden Operanden.

*) Ein assoziativer Speicher mit dreiwertiger Logik heißt im allgemeinen "funktionaler Speicher".

Adresse	Operation	"Immediate" *)	alter Selektor	neuer Selektor **)	Maske	Daten- quelle	Daten- ziel	*** Folge- adresse	Kommentar
0	search	1 0 0 1	1	P	0	-	-	1	suche Anfang Maskenfeld
1	read	0 0 0 0	P	shd P	1	-	R1	2	lies Maske für 1. Operanden
2	read	0 0 0 0	P	shd P	1	-	R2	3	lies Maske für 2. Operanden
3	search	0 - - -	1	S	0	-	-	4	suche alle Daten
4	write	- 1 0 -	S	S	0	-	-	5	setze alle "aktiv", "kein Carry"
5	read	0 0 0 0	P	shd P	1	-	M	6	lies Bitmaske für 2. Operanden
6	search	0 1 0 -	1	S	M	-	-	7	suche: aktiv, Wert 0, kein Carry
7	search	0 1 1 -	1	or S	M	R1	-	8	oder aktiv, Wert 1, Carry
8	write	0 0 0 -	S	S	0	-	-	9	setze inaktiv, Wert und Carry bleiben
9	read	0 0 0 0	P	shd P	1	-	M	10	lies Bitmaske für 1. Operanden
10	search	0 0 1 -	1	S	M	R2	-	11	suche: aktiv, Wert 1
11	write	0 0 1 -	S	S	M	R1	-	12	setze inaktiv, Wert 0, Carry
12	search	0 0 1 -	1	S	M	R1	-	13	suche: aktiv, Wert 0
13	write	0 0 0 -	S	S	M	R2	-	14	setze inaktiv, Wert 1, kein Carry
14	search	0 - - -	1	S	0	-	-	15	suche alle Daten
15	write	- 1 - -	S	S	0	-	-	16	setze alle Daten aktiv
16	read	0 0 0 0	P	shd P	1	-	M	6/17	lies neue Arbeitsmaske
17	...	.							

*) "-" bedeutet "don't care"

**) "shd" bedeutet "shift down"

***) es gibt eine Verzweigung, wenn der Inhalt von M Null ist

Abbildung 3.3: Programm zur Vektoren-Addition in einem funktionalen Speicher (nach /56/)

Bemerkungen: Im Prinzip werden hier die logischen Verknüpfungen, aus denen sich sonst der einzelne Additionsschritt aufbauen läßt - z.B. bei assoziativen Prozessoren wie dem STARAN -, durch Kombination von Abfragen nach Operandenwerten und Übertrag ersetzt. Dabei ergeben sich Vereinfachungen, weil die Summe den ersten Operanden ersetzt. Die Schleife besteht aus 11 Operationen hier, sie muß für Operanden mit je ℓ Bits ℓ Mal durchlaufen werden; eine Überlauftfeststellung enthält sie hier nicht.

Die Datenlänge ist beschränkt: Wenn wir wieder r schreiben für die Register- und Speicherwortlänge, kann höchstens sein:

$$\ell \leq \frac{r-4}{2}$$

Ein Parameter n tritt hier (außer in der Beschränkung des Speichers) nicht auf. $2\ell+3$ Speicherworte werden für die Steuerinformation gebraucht.

Laufzeitabschätzungen: $O(\ell)$ oder auch: $O(r)$.

Es läßt sich eine Verallgemeinerung angeben, wo beliebig lange Daten je in aufeinanderfolgenden Worten abgespeichert sind, auch Operanden und Ergebnisse hintereinanderstehen. Dann kommen wir zu Laufzeiten mit $O(\frac{\ell}{r})$.

3.1.2 Arithmetische Pipeline-Arrays

In verschiedenen Veröffentlichungen (/13/, /27/, /32/, /33/, /34/) werden Hardware-Komponenten für nichtlineare Grundoperationen wie Multiplikation, Division, Quadrieren und Quadratwurzelziehen beschrieben als "pipeline cellular arrays".

Solch ein Array ist aus Addierer-/Subtrahierer-Zellen und Steuer-Zellen aufgebaut. Diese Zellen sind, in einer regelmäßigen Anordnung, miteinander verknüpft und zwar haben wir eine Reihe von "Zeilen" wachsender Länge aus Addierern/Subtrahierern, pro Zeile am Rand ein Steuerelement. Es sind die Elemente innerhalb einer Zeile verbunden, etwa zur Weitergabe eines Übertrages, sowie Elemente in aufeinanderfolgenden Zeilen zur Weitergabe von

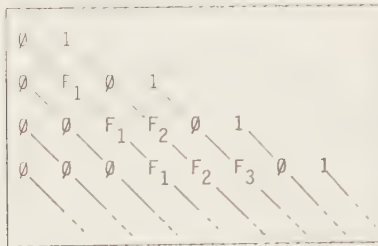
Operanden und Zwischenergebnissen. Das Array arbeitet getaktet. Die Operanden werden teils "von oben" in die Rechen-Zellen eingegeben, teils "am Rand" in die Steuer-Zeilen. Das Ergebnis erscheint, je nach Operation, "unten" an der untersten Reihe der Rechenzellen, teils "am Rand" (vgl. Abbildung 3.6).

In /24/ wird ein solches Array beschrieben, das alle vier nichtlinearen Grundoperationen ausführen kann. Ein solches Array kann als Pipeline für beliebig lange Folgen von Operanden mit beliebig wählbaren Operationen arbeiten. Dieses allgemeine Array will ich hier skizzieren.

Multiplikation und Division arbeiten mit Rechtshifts und Additionen bzw. Subtraktionen, wobei Multiplikand bzw. Divisor unverändert durch das Array weitergegeben werden. Das Quadratwurzelziehen arbeitet mit "Rekonstruktion des Restes", d.h. es wird subtrahiert, aber der alte Wert bleibt erhalten, wenn die Differenz negativ war. Das Quadrieren benutzt dasselbe Schema (Abbildung 3.4) wie das Quadratwurzelziehen zum additiven Aufbau des Ergebnisses.

Ich benutze für die folgenden Abbildungen die Bezeichnungen der Veröffentlichung /27/.

Abbildungen 35 a), b) skizzieren die beiden Typen von Zellen, Abbildung 36 gibt das Schemabild für die Zusammenschaltung.



Quadratwurzel: F_i aus der Zeile i (Vorzeichen des Restes);
Quadrat: F_i Stelle i des Operanden

Abbildung 3.4: Schema der Subtrahenden/Addenden bei Quadratwurzel/Quadrat
(nach /27/)

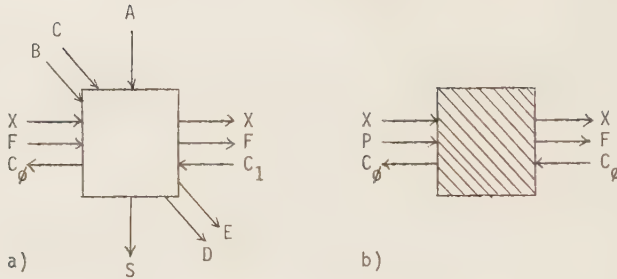


Abbildung 3.5: a) Addierer/Subtrahierer-Zelle
b) Steuer-Zelle (nach /27/)

Es sollen folgende Beziehungen gelten

a) für die Addierer-/Subtrahierer-Zelle:

$$S = (A \oplus (B \oplus X) \oplus C_1) \wedge F \vee A \wedge \neg F$$

$$C_\emptyset = (B \oplus X) \wedge (A \vee C_1) \vee A \wedge C_1$$

$$D = B \wedge C \vee C \wedge F \equiv C \wedge (B \vee F)$$

$$E = B \vee C \wedge F \equiv (B \vee C) \wedge (B \vee F)$$

b) für die Steuer-Zelle:

$$F = C_\emptyset \wedge X \vee P \wedge \neg X$$

Interpretation:

- A ist der erste Operand bei Addition/Subtraktion, wird aus der vorigen Zeile übernommen
- B ist der zweite Operand bei Addition/Subtraktion, wird aus der vorigen Zeile übernommen
- S ist das Ergebnis, geht als erster Operand an die nächste Zeile
- D geht als zweiter Operand an die nächste Zeile
- C₁ ist der eingehende Übertrag, innerhalb der Zeile
- C_∅ ist der ausgehende Übertrag, innerhalb der Zeile
- C dient zum Ändern des zweiten Operanden (für die nächste Zeile) bei Quadrat und Quadratwurzel, zum Weitergeben bei Multiplikation und Division, wird aus der vorigen Zeile übernommen
- E ist das C entsprechende an die nächste Zeile weitergegebene Signal

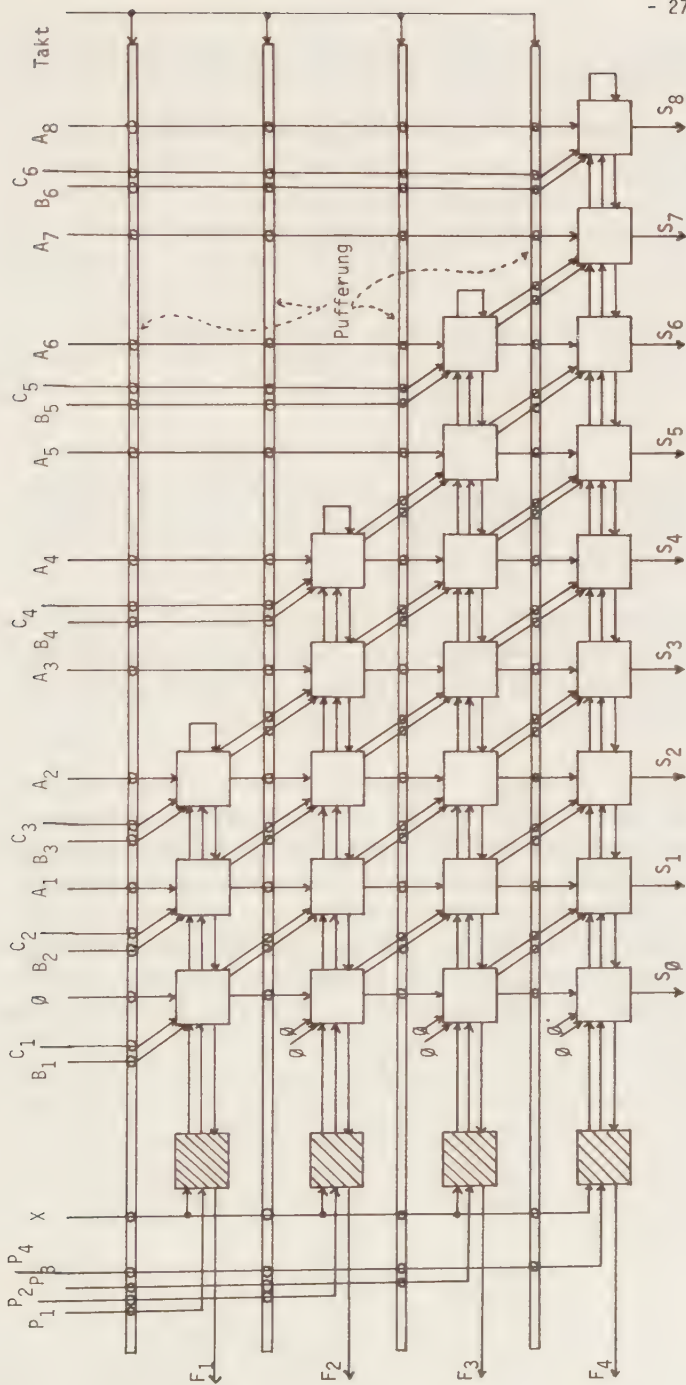


Abbildung 3.6: Schema eines verallgemeinerten Pipeline-Arrays (nach /27/)

$F_1, \dots, F_4$ Ergebnis bei Division und Quadratwurzelziehen,
 $S_0, \dots, S_8$ Ergebnis bei Multiplikation und Quadrieren, sonst: Rest

- F ist der Wert des Multiplikators (auch beim Quadrieren) oder die letzte Ergebnisstelle beim Dividieren und Wurzelziehen
- P ist der Wert des Multiplikators (auch beim Quadrieren)
- X markiert, ob addiert oder subtrahiert wird,
1: Subtraktion, d.h. Division oder Quadratwurzel,
 $\emptyset$: Addition, d.h. Multiplikation oder Quadrat

Arbeitsweise:

Multiplikation: $X = \emptyset$: B ist Multiplikand, P ist Multiplikator

Division: $X = 1$: A ist Dividend, B ist Divisor

Quadrat: $X = \emptyset$: P ist Operand

Quadratwurzel: $X = 1$: A ist Radikand

Wie man aus der Abbildung 3.6 sieht, können mit jedem Takt alle Variablen, einschließlich X, neu eingegeben werden. Die Ergebnisse $F_1, \dots, F_4$ müssen im Takt abgenommen werden (bit-seriell).

Bei großem Durchsatz können wir für jede der vier Grundoperationen von einer Laufzeit von "1 Zeiteinheit" sprechen. Sonst, das heißt: bei Einzelbetrieb, brauchen wir ℓ Zeiteinheiten, wo ℓ die Länge der Pipeline ist und die Datentlänge (in Bits) für Faktoren, Divisoren, Wurzeln.

In die "Zeiteinheit" geht implizit die "Breite" des Arrays ein, d.h. die maximale Anzahl der Zellen, nämlich als Zeit zur Fortpflanzung eines Signales quer durch eine ganze Zeile und zwar doppelt, etwa für ein Steuersignal und anschließend einen Übertrag. Diese Zeit ist $2 \cdot 2 \cdot \ell$ Hardware-Grundzeiten. Für die Division und das Quadratwurzelziehen brauchen wir, bei Implementierung von Algorithmen "mit Rekonstruktion des Restes", wie sie hier vorgeschlagen sind, $2 \cdot \ell$ zusätzliche Grundzeiten, weil das Signal F wieder zurücklaufen muß zur Entscheidung, ob die berechnete Differenz oder der alte Rest weitergegeben werden soll.

3.2 Allgemeine Überlegungen zu den mikroimplementierten Algorithmen

Allgemeines Problem: Gegeben seien n gleichformatige Datenworte oder Datenwortpaare der Länge ℓ (in Bits). Unter den Datenworten oder Datenwortpaaren wird ausgewählt durch eine Auswahlmaske m ; unter den Bits innerhalb aller Datenworte wird ausgewählt durch eine Relevanzmaske e . Zur gleichartigen Bearbeitung aller ausgewählten Datenworte eines solchen Datenbestandes haben wir einen Satz von Operationen: Berechnen eines Funktionswertes aus allen ausgewählten Datenworten. Gesucht sind effektive Algorithmen zur Implementierung solcher Operationen durch Mikroprogramme auf einem konventionellen Rechner der Speicherwortlänge bzw. Pechenwerkbreite r .

Forderung an die Algorithmen: Sie müssen *homogen* sein, das soll heißen: Sie müssen unempfindlich sein gegen signifikante Dateneigenschaften wie zum Beispiel das Vorzeichen, um alle Daten gemeinsam ohne Fallunterscheidungen, die mehrere Durchläufe erfordern würden, in einem einzigen Durchlauf des Mikroprogrammes verarbeiten zu können und nichts von der maximal möglichen Parallelität der Operation zu verlieren. Die Horizontalverarbeitung erlaubt die operanden-abhängigen Fallunterscheidungen mit Verzweigungen, bei der Vertikalverarbeitung muß man Fallunterscheidungen, wo sie nötig sind, durch logische Operationen implementieren, zum Beispiel mit speziellen "Masken".

Klassifikation der Operationen: Alle diejenigen Operationen sollen "*linear*" heißen, bei denen die Information der $n \cdot \ell$ Bits des Datenbestandes (soweit sie ausgewählt sind) genau einmal eingehen. Alle anderen Operationen, d.h. solche, bei denen die $n \cdot \ell$ Bits in mehr als einer Kombination ausgewertet werden, sollen hier "*nicht-linear*" heißen. Lineare Operationen sind zum Beispiel Vektoren-Addition, Komponentensumme, Maximumsuche. Nicht-lineare Operationen sind zum Beispiel Vektoren-Multiplikation und "Best Match"-Suche. Alle Operationen, bei denen nur Bits innerhalb eines Datenwortes bzw. Datenwortpaares miteinander verknüpft werden, sollen "*Einzeldaten-Operationen*" heißen. Alle anderen, d.h. solche, bei denen Bits verschiedener Datenworte bzw. Datenwortpaare zur Erzeugung eines Ergebnisbits ausgewertet werden müssen, sollen "*Gesamtdaten-Operationen*" heißen. Einzeldaten-Operationen sind zum Beispiel Vektoren-Addition und -Multiplikation, "Exact Match"-Suche. Gesamtdaten-

Operationen sind zum Beispiel Komponentensumme, Maximumsuche, "Best-Match"-Suche. Einzeldaten-Operationen haben eine *Verknüpfungsrichtung* und zwar längs der Datenworte. Gesamtdaten-Operationen haben meistens zwei *Verknüpfungsrichtungen*; das gilt für alle, die in unseren Befehlslisten skizziert werden. Die Pack-Operation, die in 3.4.4 beschrieben wird, wäre ein Beispiel für eine Gesamt-Operation mit einer *Verknüpfungsrichtung*.

Alle linearen und nicht-linearen Einzeldaten-Operationen lassen sich einfach als Vertikaloperationen implementieren: Einzeldaten-Operationen lassen sich durch logische Verknüpfungen implementieren, da hier nur spaltengetreu je Bits an derselben Stelle in verschiedenen Speicherworten verknüpft werden müssen. Dabei können alle Datenworte bzw. Datenwortpaare parallel verarbeitet werden mit einer "Parallelität von r ". Wir arbeiten problemangepaßt bitseriell, wort-parallel - bezogen auf die Datenworte (im Gegensatz zu Speicherworten). Die Laufzeitabschätzungen liefern, bei allgemeinen n, ℓ, r , bestmögliche Ergebnisse, was die funktionale Abhängigkeit von den Dimensionsparametern betrifft. Der Vorrat der konkret verfügbaren logischen Verknüpfungen beeinflusst die konkreten Laufzeiten (siehe hierzu auch 3.5.2.1).

Lineare und nicht-lineare Einzeldaten-Operationen erfordern im allgemeinen für die Horizontalverarbeitung spezielle Hardware-Befehle oder sind weniger effektiv als bei Vertikalverarbeitung: Die linearen arithmetischen Einzeldaten-Operationen brauchen für ein Datenwort bzw. ein Datenwortpaar eine Laufzeit von $O(\log_2 \ell)$, im wesentlichen zur Berechnung von Überträgen; die nicht-linearen arithmetischen Operationen brauchen $O(\ell \cdot \log_2 \ell)$ unter der Voraussetzung, daß sie durch logische Verknüpfungen realisiert werden. In konventionellen Rechnern heutzutage werden die arithmetischen Operationen aber durch pseudoparallele Hardware beschleunigt. Man kann davon ausgehen, daß die Taktzeit so gewählt ist, daß sie ausreicht, eine lineare arithmetische Operation auszuführen, d.h. logische Verknüpfungen und lineare arithmetische erscheinen als Operationen gleichen Aufwandes. Für eine nicht-lineare arithmetische Verknüpfung müssen wir, bei konventioneller Hardware-Unterstützung, mit $O(\ell)$ rechnen. Die Laufzeitabschätzungen müssen, mit solcher pseudoparalleler Hardware, bei allgemeinen n, ℓ, r dieselben Größenordnungen liefern wie bei Vertikalverarbeitung. Die Einzeldaten-Operationen vom Typ der Vergleichs- und

Suchoperationen können durch logische Verknüpfungen implementiert werden, sie diskriminieren aber nur in Einheiten von je r Bits, insgesamt gilt: Es wird wort-seriell gearbeitet, bit-pseudoparallel oder bit-parallel, bezogen auf die Datenworte; wir haben eine "Pseudoparallelität" oder Parallelität von r .

Bemerkung: Von der "Spezialhardware" für die Horizontalarithmetik profitiert auch die Vertikalarithmetik beim Mitzählen von Schleifendurchläufen, Adreß-erzeugen usw.

Für Gesamtdaten-Operationen gibt es keine einheitliche Strategie.

Masken "quer" zur Verarbeitungsrichtung werden wie Datenbits verknüpft und beeinflussen die Laufzeit nicht: Bei allen Einzeldaten-Operationen wird bei Vertikalverarbeitung die Auswahlmaske mit den Operanden verknüpft; bei Horizontalverarbeitung dient sie als Entscheidungskriterium, ob ein Operand bzw. Operandenpaar verarbeitet werden soll.

3.3 Notation

3.3.1 Darstellung von Zahlen und Zeichen

Seien gegeben n Worte der Länge ℓ (in Bits). Habe jeweils das höchstwertige bzw. Vorzeichenbit eines Wortes die Nummer (Bitadresse) $\emptyset$.

Horizontale Darstellung von n Festkommazahlen im Zweierkomplement $\bar{a}_{\emptyset}, \bar{a}_1, \dots, \bar{a}_{n-1}$:

$$\bar{a}_i = a_{i\emptyset} \cdot 2^{\ell-1} + (-1)^{a_{i\emptyset}} \cdot \sum_{j=1}^{\ell-1} a_{ij} 2^{\ell-1-j}, \quad i = \emptyset, 1, \dots, n-1$$

mit einer Bit-Matrix $(a_{ij}), \quad \begin{matrix} i = \emptyset, 1, \dots, n-1 \\ j = \emptyset, 1, \dots, \ell-1 \end{matrix}$

Vertikale Darstellung von n Festkommazahlen im Zweierkomplement $a'_{\emptyset}, a'_1, \dots, a'_{n-1}$:

$$a'_j = a_{\emptyset j} \cdot 2^{\ell-1} + (-1)^{a_{\emptyset j}} \cdot \sum_{i=1}^{\ell-1} a_{ij} 2^{\ell-1-i}, \quad j = \emptyset, 1, \dots, n-1$$

mit einer Bit-Matrix $(a_{ij}), \quad \begin{matrix} i = \emptyset, 1, \dots, \ell-1 \\ j = \emptyset, 1, \dots, n-1 \end{matrix}$

Die Adressierrichtung der Bits, die dieser Darstellung zugrunde liegt, erweist sich als besonders gut geeignet, auf Rechnern mit automatischer Adreßerhöhung die Ein-Operanden-Befehle (wie etwa Extremumsuche) effektiv zu implementieren.

n alphanumerische Zeichenketten der Länge ℓ (in beliebiger Codierung) werden entsprechend abgespeichert.

Die Zahlen n und ℓ sind die kritischen Größen der Laufzeitabschätzungen. Bei "vielen langen" Daten muß die Verarbeitung zusätzlich in derjenigen Richtung sequentialisiert werden, bei der die Beschränkung auf die Speicherwortlänge überschritten wird: Das betrifft horizontal die Datenlänge ℓ , vertikal die Datenmenge n. Die Bit-Matrix (a_{ij}) muß für horizontale und vertikale Darstellung unterschiedlich segmentiert werden.

Sei r die Speicherwort- und Rechenwerkbreite. Es seien abgekürzt:

$$p = \left\lceil \frac{n-1}{r} \right\rceil$$
$$q = \left\lceil \frac{\ell-1}{r} \right\rceil$$

das heißt: p bzw. q sind die kleinsten ganzen Zahlen, die größer oder gleich $\frac{n}{r}$ bzw. $\frac{\ell}{r}$ sind; sie geben an, wieviele Speicherworte man braucht zur Unterbringung von n bzw. ℓ Bits.

Für n Datenworte mit je ℓ Bits braucht man bei horizontaler Darstellung

$$n \cdot q = n \cdot \left\lceil \frac{\ell-1}{r} \right\rceil$$

Speicherworte, bei vertikaler Darstellung

$$\ell \cdot p = \ell \cdot \left\lceil \frac{n-1}{r} \right\rceil$$

Das bedeutet: Der Speicherbedarf liegt, wie zu erwarten, in derselben Größenordnung. Bei horizontaler Darstellung kann "Verschnitt" auftreten, wenn ℓ kein ganzes Vielfaches von r ist - in der Praxis wird man dann ℓ aufrunden auf ein Vielfaches von r -, bei vertikaler Darstellung tritt "Verschnitt" auf, wenn n kein ganzes Vielfaches von r ist.

Die Einzelzahlen bei horizontaler Darstellung sollen rechtsbündig gespeichert werden, das soll heißen: Das niedrigstwertige Bit einer Zahl steht im niedrigstwertigen Bit eines Speicherwortes. (Vgl. Abbildung 2.2)

3.3.2 Beschreibungssprache

Die folgenden Implementierungsansätze und Abschätzungen sind nur teilweise maschinenunabhängig. Sowohl bei den möglichen logischen Mikrobefehlen wie bei den arithmetischen Mikrobefehlen haben die Maschinen der Interdata-Familie (und damit das System IBM/360) Pate gestanden. Insbesondere sollte auf eine "aktuelle" Maschine aus der Leistungsklasse "Minirechner" zurückgegriffen werden und nicht auf eine Maschine der davorliegenden Generation mit der damals noch primitiveren Hardware und den notwendig längeren Mikroprogrammen (z.B. mikroprogrammierter Multiplikation). Ganz bewußt an die Interdata 85 schließe ich mich an bei der Berücksichtigung asynchron mit dem Mikroprogramm überlappender Speicherzugriffe. Da es mir dabei im wesentlichen auf diese teilweise Überlappung ankommt, habe ich die Mikrooperationen zur Vorbereitung der Speicherzugriffe (Setzen des Adreßregisters zum Beispiel) usw. vernachlässigt. Auf jeden Fall sollte vermieden werden, hier konkrete Mikroprogramme eines konkreten Rechners aufzuschreiben, etwa in dessen Mikroassembler-Sprache. Stattdessen wurde eine höhere Sprache entworfen, vom Algol-Typ, die es erlaubt, "räumliche" und "zeitliche" Wiederholungen darzustellen sowie die erwähnten asynchronen Überlappungen. Dabei bleiben viele Details der Buchführung bei konkreten Mikroprogrammen unberücksichtigt; man kann aber sowohl die Größenordnung von Laufzeiten als Funktionen von wesentlichen Parametern abschätzen wie auch von dem Programmier- und Speicheraufwand der skizzierten Mikroprogramm-Algorithmen sich ein Bild machen.

Im folgenden sollen die *wesentlichen Sprachelemente* beschrieben werden.

```
1. sequ  $i = n-1, n-2, \dots, \emptyset$   
    $\vdots$   
end sequ
```

Die Sequenz der Anweisungen, die hier durch die drei senkrecht untereinanderstehenden Punkte abgekürzt ist, wird nacheinander für die angegebenen Indexwerte ausgeführt (die, wie üblich, nur zum Zählen dienen können oder explizit in den Anweisungen auftreten). Wir haben hier die übliche Schleife.

```
2. par  $j = 1, 2, \dots, \ell$   
    $\vdots$   
end par
```

In allen Anweisungen, die durch die drei senkrecht untereinanderstehenden Punkte symbolisiert sind, werden die angegebenen Operationen gleichzeitig

(synchron) für alle angegebenen Indexwerte ausgeführt. Diese Möglichkeit, parametrisierte Parallelausführung zu beschreiben, fehlt in allen allgemein bekannten höheren Programmiersprachen. L. Lamport /30/ gibt eine spezielle Fortran-Erweiterung an, die die Parallelisierung von zunächst sequentiellen DO-Schleifen erlaubt und zwar sowohl für asynchrone Ausführung auf einer Multiprozessor-Anlage wie auch für synchrone Ausführung auf einem Feldrechner (zum Beispiel ILLIAC IV).

```
3. sim      :  
      :  
      with   :  
      :  
      with   : (beliebig viele with : )  
      :  
      end sim
```

Alle Sequenzen, die hier durch je drei untereinanderstehende Punkte abgekürzt sind, werden gleichzeitig begonnen und überlappen sich asynchron; nach end sim sind alle diese Sequenzen fertig (end sim dient also zum Synchronisieren). Diese Konstruktion entspricht der "collateral Clause" in Algol 68.

Da bei der hier betrachteten konventionellen Hardware die Möglichkeiten zur (asynchronen) Überlappung begrenzt sind, soll als Normalfall die sequentielle Ausführung sequentiell aufgeschriebener Anweisungen angenommen werden. Als Begrenzer werden Semikolons benutzt.

```
4. if ...  
      then :  
      :  
      else :  
      :  
      end if
```

Als übliche Zweifachverzweigung

Soll eine der beiden Alternativen die Ausführung einer Iterationsschleife beenden, wird sie mit exit markiert; soll der Sprung aus k ineinandergeschachtelten Schleifen herausführen, wird exit k geschrieben.

5. Doppelpunkt mit Gleichheitszeichen wird als Symbol für die Zuweisung benutzt.

6. +, - bedeuten Addition und Subtraktion (zum Beispiel auch für die Indizes); $\wedge$, $\vee$, $\oplus$ bedeuten die binären logischen Verknüpfungen Und, Oder, Exklusives Oder, ist die logische Negation.

read, *write* sind Speicherzugriffe.

Bei Bedarf werden weitere Operationssymbole benutzt wie zum Beispiel *addsub* ($a_{i,j}$, $b_{i,j}$); sie werden an Ort und Stelle definiert bzw. es wird auf ihre Definition verwiesen.

7. *equal*, *unequal*, *greater*, *notgreater* usw. bedeuten die üblichen Relationen.
8. Es gibt, außer den Indizes, keine lokalen Variablen. Indizes können (für Laufschleifen) aufwärts und abwärts gezählt werden mit beliebiger Schrittweite.
9. Es gibt keine expliziten Deklarationen. Implizit (das heißt: durch Konvention) ist deklariert:
Indizierte oder nicht-indizierte Variable sind Bit-Variable; Variable mit einem Querstrich sind Festkommazahlen-Variable; $\bar{a}_i^{(k)}$ ist das k-te Teilwort der Festkommazahlen-Variablen $\bar{a}_i$.
10. $\emptyset$, 1 sind binäre Konstanten; *null*, $\bar{1}$, $\bar{2}$ usw. sind Festkommazahlenkonstanten; bei den Indizes ist der Querstrich weggelassen.
11. Kommentar kann an beliebigen Stellen stehen (außer innerhalb einfacher Anweisungen), er wird durch Anführungszeichen markiert.

Die Sprachsymbole werden durch Unterstreichung hervorgehoben. Es wird ein Satzspiegel gewählt, der die Kontrollstruktur erkennen läßt.

3.4 Befehlslisten für einzelne Anwendungsklassen. Implementierungsskizzen

Es erscheint sinnvoll, die Befehlsliste mehr unter dem Aspekt der möglichen Anwendungsklassen zu entwerfen als mit Blick auf die imitierten speziellen Hardware-Systeme; vor allem, weil uns die große Flexibilität der mikropro-

grammierten Implementierung gelegentlich erlaubt, die Befehlsliste für eine Anwendungsklasse besser abzurunden, als dies mit hochspezialisierter Hardware möglich wäre. Es werden drei spezielle Anwendungsklassen betrachtet: Vektoren-Arithmetik, assoziative Verarbeitung, Muster-Verarbeitung. Als Ergänzung zu allen drei Befehlslisten ist eine Liste von Hilfsoperationen angegeben, die für alle drei Anwendungsklassen mehr oder weniger notwendig sind. Eine solche Gliederung in Befehle für verschiedene Anwendungsklassen ist auch aus praktischen Gründen notwendig: Die Erfahrungen mit der Pilot-Implementierung legen es nahe, Mikroprogramm-Moduln zu bilden und wahlweise zu laden, weil für das gesamte Mikroprogramm-Paket der Speicher nicht ausreichen wird.

Zu jeder der Anwendungsklassen wird eine Liste von Befehlen angegeben, die notwendig ist, um Benutzer-Algorithmen für diese Anwendungsklasse konsistent und effektiv formulieren (programmieren) zu können. Dabei zeigt sich gelegentlich (z.B. bei der Gleitkomma-Arithmetik), daß manche Operationen mit realisiertischem Aufwand nicht vertikal implementierbar sind, jedenfalls nicht im allgemeinen Ansatz.

Zu den einzelnen Befehlen werden ein oder mehrere Implementierungs-Ansätze angegeben, mit Laufzeit-Abschätzungen als Funktionen von datenbestimmenden Parametern wie Anzahl der Operanden und Zahlengenaugkeit. Der allgemeine Fall (n und ℓ beliebig) ist an einigen Stellen in den formalen Beschreibungen explizit durchgeführt, in den anderen in den Abschätzungen erwähnt. In den Abschätzungen ist, wo es möglich war, r wie ein Parameter berücksichtigt, um Aussagen zu gewinnen über günstige Speicherwortlängen.

Die hier angegebenen Abschätzungen sind Grenzwertabschätzungen für große Parameterwerte. In Abschnitt 3.5.1 sind konkrete Werte zusammengestellt, die durch Auszählen der Mikrobefehle gewonnen wurden, um die verschiedenen Ansätze für reale Anwendungsbeispiele vergleichen zu können.

Nicht berücksichtigt wird in den formalen Beschreibungen der Mechanismus, wie die horizontalen oder vertikalen Komplexoperationen aufgerufen werden, z.B. also auch, wie die Dimensionsparameter übernommen werden.

Bemerkung: Es handelt sich bei den skizzierten Befehlen um einzelne "Assembler-befehle", die in geeigneter Form in einer höheren Programmiersprache wie zum Beispiel der Algol 68-Erweiterung von B. Jahn /25/ aufrufbar sein sollen.

3.4.1 Vektoren-Arithmetik

Es geht hier weniger um Vektorkalkül im Sinne der Mathematik als um - bei "makroskopischer" Betrachtung - parallele gleichartige Arbeit, auf mehreren gleichartigen numerischen Daten. Der Vektorkalkül im eigentlichen Sinn kann dann ein geeigneter Anwendungsbereich für diese Arithmetik sein, vgl. Kapitel 4.

Das Prinzip der Vertikalverarbeitung besteht darin, stellengleiche gleich-lange Spalten (in verschiedenen Teilmatrizen) zu verknüpfen zu Ergebnisspalten an entsprechenden Stellen der Ergebnis-Teilmatrix. Sind die Spalten Binärcodierungen von Festkommazahlen, so stellt eine Teilmatrix einen Vektor mit r Komponenten dar, und die skizzierte Verknüpfung ist eine komponentenweise Vektor-Vektor-Verknüpfung. Dabei ist die Anzahl r der Komponenten beliebig wählbar, kann also zum Beispiel von Befehl zu Befehl variieren. Abbildung 3.7 gibt ein Beispiel für eine Vektoren-Addition/Subtraktion.

Bei der Untersuchung von konkreten Beispielen findet man, daß sehr oft für einen der beiden Operanden in allen Spalten derselbe Wert gebraucht wird, den man aber nicht gerne vorher entsprechend oft reproduzieren möchte. Es werden also mehrere Zahlen (in verschiedenen Spalten) mit einer einzigen Zahl verknüpft. Diese Verknüpfungen wollen wir "Vektor-Skalar" nennen. In der Pilot-Implementierung der Vertikalverarbeitung kann frei gewählt werden ob der "Skalar" in konventioneller (horizontaler) Form vorgegeben ist oder selbst Komponente eines Vektors; mit einem Orthogonalspeicher ist eine solche Unterscheidung unwichtig.

Bei anderen Beispielen (etwa der Schnellen Fourier-Transformation, vgl Kapitel 4) tritt aber auch die Aufgabe auf, zwar gleichviele Komponenten zweier Operandenvektoren und in relativ gleicher Anordnung zu verknüpfen, aber nicht mehr stellengleich. Prinzipiell läßt sich diese Aufgabe mit zusätzlichen Schiebeoperationen innerhalb der Verknüpfungsschleifen oder auch Zusatz-Hardware wie den "Quersummen-Modul" (siehe 3.5.2.2) lösen. Ob solche Verknüpfungen noch konsistent zur "Vektoren-Arithmetik" gehören, muß danach entschieden werden, welche Aufgaben gelöst werden sollen. Bei der horizontalen (konventionellen) Emulation treten hier keine Probleme auf, ebensowenig wie bei den "Vektor-Skalar"-Verknüpfungen.

Einmal, um bei Vertikalverarbeitung auch mit Vektoren von weniger als r Komponenten den Speicher einigermaßen nutzen zu können, zum anderen, um auf den Komponenten eines Vektors etwa abhängig von Bedingungen verschieden arbeiten zu können, haben wir eine "Selektionsmaske" (auch "Aktivitätsmaske" genannt wie bei den assoziativen Operationen, vgl. 3.4.2). Das ist ein Bitvektor mit r Komponenten; eine Komponente dieses Bitvektors ist 1, wenn auf der entsprechenden Komponente des Zahlenvektors gearbeitet werden soll. Prinzipiell gilt dieselbe Selektionsmaske für beide Operanden und das Ergebnis, da wir ja stellengleich verknüpfen wollen. Die Vertikalverarbeitung erlaubt, diese Maske sehr einfach zu speichern und zu verarbeiten; bei Horizontalverarbeitung ist es schwieriger, eine passende Darstellung der Masken zu finden.

Bisher sind wir davon ausgegangen, daß Operanden und Ergebnisse gleich lang sind. Ausnahme: Produkt bei Multiplikation und Dividend bei Division sind jeweils doppelt so lang wie die anderen Zahlen bei diesen Operationen, entsprechend: Radikand beim Wurzelziehen, verglichen mit dem Ergebnis. Das braucht eigentlich nicht so zu sein; aber eine Flexibilität des Verhältnisses der Zahlenlängen kostet zusätzlichen Mikroprogramm-Aufwand.

Die zugrundeliegenden Zahlen (Werte der Vektoren-Komponenten) sind binäre Festkommazahlen im Zweierkomplement. Gleitkommazahlen zu verarbeiten bedeutet vor allem bei Addition und Subtraktion einen enormen Aufwand für die notwendige Adjustierung. Für ein ausschließlich vertikal arbeitendes System ist dieser Aufwand nicht zu rechtfertigen. Hat man einen Orthogonalspeicher, kann man die Adjustierung für alle bis zu n Operandenpaare vorab ausführen durch konventionell benutzte Shift- und Subtraktionsbefehle. Anschließend könnte dann die eigentliche Verknüpfung der Mantissen wieder als Vektorenverknüpfung ausgeführt werden, was sich aber vermutlich nicht lohnt. Bei vergleichbaren Systemen mit autonom arbeitenden 1-Bit-Prozessoren wie DAP kann Gleitkomma-Arithmetik effektiv implementiert werden. In der Literatur findet sich gelegentlich ein Konzept, das die Schwierigkeiten beim Adjustieren umgeht: Unter der Voraussetzung, daß alle Zahlen eines Vektors ungefähr in derselben Größenordnung liegen, benutzt man für alle Komponenten eines Vektors denselben Exponenten. Es soll jeweils mindestens eine der Komponenten normiert sein. Der Exponent wird außerhalb des Vektors, davor oder dahinter oder sonst an eindeutiger Stelle, gespeichert, etwa in Horizontaldarstellung. Die Einschränkung auf einen gemeinsamen Exponenten bedeutet aber natürlich im wesentlichen: Festkommazahlen mit "variablen" Komma.

Die Befehlsliste:

1. Addition, komponentenweise und mit gemeinsamem "Skalar",
2. Subtraktion, entsprechend der Addition,
3. Berechnung der Zweierkomplemente,
4. Multiplikation, komponentenweise und mit gemeinsamem "Skalar",
5. Division, Entsprechend der Multiplikation,
6. Berechnen der Quadratwurzeln,
7. Summation von Komponenten eines Vektors,
8. Komponentenweiser Vergleich von Vektoren,
9. Suche nach dem arithmetischen Maximum unter den Komponenten eines Vektors,
10. Suche nach dem arithmetischen Minimum unter den Komponenten eines Vektors,
11. Schwellwertsuche: Markierung, welche Komponenten eines Vektors kleiner, gleich, größer einem vorgegebenen Wert sind,
12. Zählen addierend/subtrahierend, d.h. Addition/Subtraktion des Wertes 1 in allen markierten Komponenten.

Dabei werden zwar 1., 2., 4., 5., 8. auch als "Vektor-Skalar"-Operationen gebraucht, in den folgenden formalen Beschreibungen werden aber für alle Befehle nur die "Vektor-Vektor"-Operationen berücksichtigt.

Konvention:

$a_{i,j}$, $b_{i,j}$ sind Bits der Operanden,
 $s_{i,j}$ sind Bits der Ergebnisse,
 m_i bzw. m_j sind Bits der Selektionsmaske,
 o_i bzw. o_j sind Bits der Überlauffeststellung,
 d_j sind Bits der Operationencodierung,
 $x_{i,j}$, y_j usw. sind Bits von Zwischenwerten,
 $c_{i,j}$ sind Übertragbits.

Ich habe einbuchstabile Identifikatoren gewählt in Ablehnung an in der Hardware geläufige Bezeichnungen für Busse, Signale usw.

Eine vollständige Liste der Identifikatoren in den Mikroprogramm-Algorithmen findet sich auf den Seiten 166 und 167.

0	0	0	1	0	1	0	0
0	0	0	1	0	1	1	1
0	1	1	1	0	1	0	1
0	0	0	1	0	1	0	1
0	1	0	1	0	1	0	1
0	5	4	-1	0	-1	(8)	(15)

1. Operandenvektor

0	0	1	1	0	0	0	0
0	1	1	1	1	1	0	0
1	0	0	0	0	0	0	0
0	1	0	0	0	1	1	0
0	0	1	1	0	0	1	0
4	10	-7	-7	8	10	(3)	(0)

2. Operandenvektor

1	1	1	1	1	1	0	0
---	---	---	---	---	---	---	---

Selektionsmaske

0	0	1	1	0	0	0	0
---	---	---	---	---	---	---	---

Operationencodierung

0	0	0	0	1	1	0	0
1	1	1	0	1	1	0	0
1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0
1	1	0	0	0	1	0	0
(15)	(9)	(14)	(0)	(-2)	(-3)	(2)	(0)

Summenvektor
vor Ausführung der Operation

0	0	0	0	0	0	0	0
0	1	1	0	1	1	0	0
1	1	0	1	0	0	0	0
0	1	1	1	0	0	1	0
0	1	1	0	0	1	0	0
4	15	11	6	8	9	(2)	(0)

Summenvektor
nach Ausführung der Operation

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

Überlaufmarkierung

Abbildung 3.7: Beispiel einer vertikalen Vektoren-Addition/Subtraktion

Addition der positiven Komponenten des zweiten Operanden, Subtraktion der negativen Komponenten des zweiten Operanden. Berücksichtigung einer Selektionsmaske. Irrelevante Information, die nicht in die Verknüpfung eingeht bzw. nicht verändert wird, ist in der Dezimaldarstellung eingeklammert.

3.4.1.1 Addition/Subtraktion

Die Implementierungsskizze für die konventionelle Emulation einer Vektoren-Addition, bedingt für die einzelnen Komponenten, findet sich in Anhang A. Die Überlauftfeststellung ist bei horizontaler Zahlendarstellung nur interessant, wenn ℓ ein ganzes Vielfaches von r ist.

Bei der Vertikalverarbeitung lassen sich Addition und Subtraktion zusammenfassen zu einem *gemeinsamen* Mikroprogramm. Durch eine Maske $\bar{d}$ wird eine Operationencodierung eingeführt: $d_j = 0$ bedeutet "Addition in Spalte j ", $d_j = 1$ bedeutet "Subtraktion in Spalte j ". Damit spart man nicht nur Speicherplatz für die Mikroprogramme, sondern gewinnt ein elegantes Werkzeug für den Booth'schen Multiplikationsalgorithmus und Algorithmen ohne Wiederherstellen des Restes für Division und Quadratwurzelberechnung.

Zunächst die Implementierungsskizze für Vertikalverarbeitung auf der *Standard-Hardware* vom Typ der Interdata 85.

sequ $k = 0, 1, \dots, p-1$

"Schleife über die Teilvektoren mit je r Komponenten"

par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerks"

$c_{\ell-1,j} := 0;$

$z_j := m_{kr+j};$

sequ $i = \ell-1, \ell-2, \dots, 0$

"Schleife über die Bits der Zahlendarstellung"

read $b_{i,kr+j};$

sim *read* $a_{i,kr+j}$

with $x_j := b_{i,kr+j} \oplus c_{i,j}$

end sim;

"lesen Operanden, beginnen Verknüpfung"

sim read $s_{i,kr+j}$

with $y_j := a_{i,kr+j} \oplus x_j;$

$y_j := y_j \wedge m_{kr+j}$

end sim;

"lesen alten Wert des Ergebnisbereiches, fortsetzen Verknüpfung"

$s_{i,kr+j} := s_{i,kr+j} \wedge z_j;$

$s_{i,kr+j} := s_{i,kr+j} \vee y_j;$

"fertig Ergebnisstelle, eingeblendet in alten Wert"

sim write $s_{i,kr+j}$

with $y_j := a_{i,kr+j} \oplus d_{kr+j};$

$y_j := x_j \wedge y_j;$

$x_j := b_{i,kr+j} \wedge c_{i,j};$

$c_{i-1,j} := y_j \vee x_j$

end sim

"schreiben Ergebnis, berechnen Übertrag"

end sequ;

"berechnen Überlauffeststellung"

$x_j := b_{0,kr+j} \oplus d_{kr+j};$

$x_j := a_{0,kr+j} \wedge x_j;$

$y_j := \neg s_{0,kr+j};$

$x_j := x_j \wedge y_j;$

$y_j := \neg b_{0,kr+j};$

$y_j := y_j \oplus d_{kr+j};$

$z_j := \neg a_{0,kr+j};$

$z_j := z_j \wedge y_j;$

$z_j := z_j \wedge s_{0,kr+j};$

$y_j := x_j \vee z_j;$

$o_j := y_j \wedge m_{kr+j};$

write o_j

end par

end sequ

Bemerkung: Die Speicherzugriffe können hier nahezu vollständig mit Verknüpfungsoperationen überlappt werden. Es ist pro Bit-Slice (pro innerste Schleife) ein Speicherzugriff mehr nötig als bei der horizontalen Blockoperation, weil der ursprüngliche Wert für die nicht ausgewählten Komponenten erhalten bleiben soll.

Die Laufzeit liegt zwar mit $O(\frac{n}{r} \cdot \ell)$ in derselben Größenordnung wie bei der horizontalen Blockoperation, aber statt einer einzigen Verknüpfungsoperation dort muß hier eine Sequenz von 9 Operationen (außer den Speicherzugriffen) ausgeführt werden. Deshalb ist Unterstützung durch Zusatz-Hardware zu wünschen. Mit dem "Vertikal-Modul" (vgl. 3.5.2.1) reduziert sich das Mikroprogramm in Speicherplatz und konkreter Laufzeit erheblich:

sequ $k = \emptyset, 1, \dots, p-1$

"Schleife über die Teilvektoren mit je r Komponenten"

par $j = \emptyset, 1, \dots, r-1$

"Parallelarbeit des Speichers bzw. Rechenwerks"

load register (null, d_{kr+j}); *)

load mask (m_{kr+j}); *)

"vorbesetzen Hilfsregister im Zusatz-Modul"

sequ $i = \ell-1, \ell-2, \dots, \emptyset$

"Schleife über die Bits der Zahlendarstellung"

read $a_{i,kr+j}$;

read $b_{i,kr+j}$;

"lesen Operanden"

sim *read $s_{i,kr+j}$*

with $x_j := \text{addsub} (a_{i,kr+j}, b_{i,kr+j})$ *)

end sim;

"lesen alten Ergebnisbereich, bilden Verknüpfung"

$s_{i,kr+j} := \text{mask} (s_{i,kr+j}, x_j);$ *)

write $s_{i,kr+j}$

"maskieren Ergebnis, schreiben"

*) genaue Beschreibung in 3.5.2.1

end sequ;

$o_j := \text{overflow}(a_{\emptyset, kr+j}, b_{\emptyset, kr+j});$ *

$o_j := \text{mask}(\underline{\text{null}}, o_j);$ *

write o_j

"berechnen Überlauffeststellung, maskieren, schreiben"

end par

end sequ

3.4.1.2 Zweierkomplement

Die horizontale Blockoperation muß aus Subtraktionen von Null implementiert werden. Die Vertikaloperation könnte analog aufgebaut werden. Es zeigt sich aber, daß es einen gemeinsamen Algorithmus gibt zur Berechnung von Zweier- und Einerkomplement. Wir benutzen auch hier eine Maske zur Operationencodierung: $d_j = \emptyset$ bedeutet "Zweierkomplement in Spalte j", $d_j = 1$ bedeutet "Einerkomplement in Spalte j". Das Einerkomplement wird zwar in der Arithmetik hier nicht gebraucht - bei den Verknüpfungen würden sich wegen der Einer-Rückläufe zusätzliche $O(l)$ Verarbeitungsschritte gegenüber Addition/Subtraktion im Zweierkomplement ergeben -, könnte aber zur Negation binärer Muster benützt werden (vgl. 3.4.3).

sequ $k = \emptyset, 1, \dots, p-1$

"Schleife über die Teilvektoren mit je r Komponenten"

par $j = \emptyset, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerks"

$z_j := \neg m_{kr+j};$

$c_{l-1,j} := d_{kr+j} \wedge m_{kr+j};$

"für Zweierkomplement wird der Übertrag mit null vorbesetzt, für Einerkomplement wird der Übertrag gleich der Auswahlmaske gesetzt"

sequ $i = l-1, l-2, \dots, \emptyset$

"Schleife über die Bits der Zahlendarstellung"

read $a_{i, kr+j};$

*) genaue Beschreibung in 3.5.2.1

sim read $s_{i,kr+j}$

with $x_j := a_{i,kr+j} \oplus c_{i,j};$

$x_j := x_j \wedge m_{kr+j};$

$y_j := a_{i,kr+j} \wedge m_{kr+j};$

$c_{i-1,j} := c_{i,j} \vee y_j$

end sim;

"lesen alten Ergebnisbereich, berechnen Komplement,
berechnen Übertrag"

$s_{i,kr+j} := s_{i,kr+j} \wedge z_j;$

$s_{i,kr+j} := s_{i,kr+j} \vee x_j;$

write $s_{i,kr+j}$

"zusammensetzen altes und neues Ergebnis, schreiben"

end sequ

end par

end sequ

Bemerkung: Auch hier kann die Sequenz der Mikrobefehle in der innersten Schleife durch einen Spezialbefehl an die Zusatz-Hardware abgekürzt werden. Laufzeitabschätzung für Vertikalverarbeitung und horizontale Blockoperation $O(\frac{n \cdot \ell}{r})$.

3.4.1.3 Multiplikation

In Anhang A ist die horizontale Blockoperation für mehrfach-genaue Arithmetik angegeben. Obwohl der dabei verwandte Algorithmus wie auch der Booth'sche Multiplikationsalgorithmus, der für die eine Vertikalimplementierung benutzt wird, erlauben, das Produkt korrekt auf den alten Wert des Ergebnisses aufzuaddieren, gehen wir hier davon aus, daß der Ergebnisbereich vorher auf null gelöscht worden ist.

Beim Booth'schen Algorithmus läßt sich sehr elegant der gemeinsame Algorithmus für Addition und Subtraktion ausnutzen (als Unterprogramm - hier aber als Sequenz des Hauptprogrammes geschrieben). Ich setze dabei bereits die Unterstützung durch Zusatz-Hardware voraus ("Vertikal-Modul" - vgl. 3.5.2.1).

sequ $k = \emptyset, 1, \dots, p-1$

"Schleife über die Teilvektoren mit je r Komponenten"

par $j = \emptyset, 1, \dots, r-1$

"Parallelarbeit des Speichers bzw. Rechenwerks"

$d_j := \emptyset;$

"Anfangswert für Operationenmaske"

sequ $i = l-1, l-2, \dots, \emptyset$

"Schleife über die Bits der Multiplikatoren"

$\text{read } b_{i,kr+j};$

"lesen Multiplikatorenstelle"

$x_j := b_{i,kr+j} \oplus d_j;$

$d_j := b_{i,kr+j};$

$x_j := x_j \wedge m_j;$

" $\bar{x}$ liefert die Information, ob im Multiplikator Bitwechsel aufgetreten ist; $\bar{d}$ enthält die aktuellen Multiplikatorenbits und damit unmittelbar die Operationencodierung"

if $\bar{x} \text{ unequal null}$

"Abfrage, ob mindestens ein Bitwechsel in den ausgewählten Komponenten"

then $\text{loadregister}(\text{null}, d_j);$

*)

$\text{loadmask}(x_j);$

*)

"Vorbesetzen Hilfsregister im Zusatz-Modul; für den aktuellen Rechenschritt wird nur auf Komponenten mit Bitwechsel gearbeitet"

sequ $h = l-1, l-2, \dots, \emptyset$

"Schleife über die Bits der Multiplikanden"

$\text{read } s_{i+h+1, kr+j};$

$\text{read } a_{h,kr+j};$

"lesen Bits des bisherigen Zwischenergebnisses und des Multiplikanden"

*) genaue Beschreibung in 3.5.2.1

```

yj := addsub (si+h+1, kr+j, ah, kr+j); *)
si+h+1, kr+j := mask (si+h+1, kr+j, yj); *)
"addieren/subtrahieren Multiplikandenstelle,
zusammensetzen Ergebnis"

write si+h+1, kr+j

```

end sequ;

sequ h = i, i-1, ..., 0

"Schleife über die oberen Stellen des Teilergebnisses
zum Vorziehen der Überträge"

```

read sh, kr+j;

yj := addsub (sh, kr+j, null); *)
sh, kr+j := mask (sh, kr+j, yj); *)
"addieren/subtrahieren Übertrag, zusammensetzen
Ergebnis"

write sh, kr+j

```

end sequ

else

end if

end sequ

end par

end sequ

Der Booth'sche Algorithmus ist stark datenabhängig: Nur bei Wechsel in der Bitfolge des Multiplikators wird addiert oder subtrahiert. Da aber diese beiden Operationen einander immer abwechseln, müssen immer bis in die Vorzeichenstellen die Überträge fortgepflanzt werden. Das bedeutet im schlimmsten Fall

$$\sum_{i=0}^{\ell-1} \ell + (\ell-i) = \ell^2 + \frac{\ell(\ell+1)}{2}$$

Additions-/Subtraktionsschritte. Insgesamt haben wir also eine Laufzeit von $O(\frac{n}{r} \cdot \ell^2)$.

*) genaue Beschreibung in 3.5.2.1

Mit dem "einfachen" Multiplikationsalgorithmus ersparen wir uns zwar die Übertragsfortpflanzung jeweils bis in die Vorzeichenstelle, brauchen aber kompliziertere Einzelschritte, um positive und negative Faktoren in beliebiger Kombination verarbeiten zu können:

sequ $k = 0, 1, \dots, p-1$

"Schleife über die Teilvektoren mit je r Komponenten"

par $j = 0, 1, \dots, r-1$

"Parallelität des Speichers bzw. Rechenwerks"

$loadmask(m_{kr+j});$ *

"vorbesetzen Auswahlmaske"

sequ $i = l-1, l-2, \dots, 0$

"Schleife über die Bits der Multiplikatoren ohne Vorzeichen"

$read\ b_{i,kr+j};$

if $b_i^{(k)} \text{ unequal } null$

then $loadregister(null, b_{i,kr+j});$ *

"vorbesetzen weitere Hilfsregister des Zusatz-Moduls"

sequ $h = l-1, l-2, \dots, 1$

"Schleife über die Bits der Multiplikanden ohne Vorzeichen"

$read\ s_{i+h+1,kr+j};$

$read\ a_{h,kr+j};$

"lesen Bits des bisherigen Zwischenergebnisses und des Multiplikanden"

$y_j := multadd(s_{i+h+1,kr+j}, a_{h,kr+j});$ *)

$s_{i+h+1,kr+j} := mask(s_{i+h+1,kr+j}, y_j);$ *)

"spezielle Addition des Multiplikanden auf das bisherige Teilergebnis (Addition genau, wenn Multiplikatorstelle den Wert 1 hat); maskieren Ergebnisstelle"

$write\ s_{i+h+1,kr+j}$

end sequ

else

end if;

*) genaue Beschreibung in 3.5.2.1

read $s_{i+1,kr+j}$;

read $a_{\emptyset,kr+j}$;

"lesen Vorzeichenstelle des Multiplikanden, entsprechende Stelle des Zwischenergebnisses"

$y_j := \text{multsign}(s_{i+1,kr+j}, a_{\emptyset,kr+j});$ *)

$s_{i+1,kr+j} := \text{mask}(s_{i+1,kr+j}, y_j);$ *)

"letzter Additionsschritt (beim Aufaddieren der Vorzeichenstelle des Multiplikanden), leistet Korrektur für negative Multiplikanden, maskieren Ergebnisstelle"

write $s_{i+1,kr+j}$

end sequ;

read $b_{\emptyset,kr+j}$;

"lesen Vorzeichenstelle des Multiplikators"

loadregister (null, $b_{\emptyset,kr+j}$); *)

"vorbesetzen Hilfsregister des Zusatzmoduls"

read $s_{l+1,kr+j}$;

read $a_{l-1,kr+j}$;

"lesen unterste Stelle des Multiplikanden und entsprechende Stelle des Zwischenergebnisses"

$y_j := \text{multleast}(s_{l+1,kr+j}, a_{l-1,kr+j});$ *)

$s_{l+1,kr+j} := \text{mask}(s_{l+1,kr+j}, y_j);$ *)

"spezieller Additionsschritt beim Aufaddieren (Subtrahieren) des Multiplikanden auf das Zwischenergebnis, enthält Addition einer 1 zur dynamischen Erzeugung des Zweierkomplementes, maskieren Ergebnisstelle"

write $s_{l+1,kr+j}$;

sequ h = l-2, l-3, ..., 0

"Schleife über die Bits der Multiplikanden ohne niedrigstwertige Stelle"

read $s_{h+2,kr+j}$;

read $a_{h,kr+j}$;

"lesen Multiplikandenstelle und Stelle des Zwischenergebnisses"

*) genaue Beschreibung in 3.5.2.1

```

yj      := multsub (sh+2,kr+j, ah,kr+j);          *)
sh+2,kr+j := mask (sh+2,kr+j, yj);              *)
"subtrahieren Multiplikanden (wo Multiplikator negativ);
maskieren Ergebnisstelle"

write sh+2,kr+j

end sequ;

sim read s1,kr+j
with yj := equivalence (a0,kr+j, b0,kr+j)          *)
end sim;
"feststellen Vorzeichenwert des Ergebnisses"

s1,kr+j := mask (s1,kr+j, yj);
write s1,kr+j;
read s0,kr+j;
s0,kr+j := mask (s0,kr+j, yj);
write s0,kr+j
"schreiben Vorzeichenwert in die beiden höchstwertigen Stellen
des Ergebnisses"

end par

end sequ
```

Bei diesem Algorithmus wird bei Multiplikatorstellen mit dem Wert 1 addiert, sonst keine Operation ausgeführt. Die Anzahl der Additionseinzelschritte ist im ungünstigsten Fall n^2 . Dafür ist das Programm statisch umfangreicher, d.h. es benötigt mehr Speicherplatz. Außerdem braucht man vier zusätzliche Spezialoperationen des "Vertikal-Moduls" bzw. mit der Standard-Hardware vier zusätzliche Unterprogramme.

*) genaue Beschreibung in 3.5.2.1

3.4.1.4 Division

Für die horizontale Blockoperation ergibt sich gegenüber der Multiplikation nichts qualitativ Neues. Für die Vertikaloperation steht mit dem "nichtzurückschreibenden" Algorithmus analog dem Booth'schen Algorithmus wieder ein Algorithmus zur Verfügung, der ganz *homogen* abläuft bei allen möglichen Vorzeichenkombinationen der Operanden. Es handelt sich um denjenigen Divisionsalgorithmus, bei dem dann, wenn Dividend (bzw. in den weiteren Schritten: Rest) und Divisor gleiches Vorzeichen haben, subtrahiert wird, sonst addiert; wenn dann der neue Rest und der Divisor dasselbe Vorzeichen haben, hat die berechnete Quotientenstelle den Wert 1, sonst hat sie den Wert 0. Man kann also die zuletzt berechnete Quotientenstelle unmittelbar als Operationenmarkierung für die Addition/Subtraktion benutzen. Etwas Entsprechendes gilt für die Korrektur des Restes. Laufzeitabschätzung: ℓ^2 Einzelschritte, im wesentlichen vom Aufwand des einzelnen Additions-/Subtraktions-Schrittes, also Laufzeitverhalten etwa wie beim "einfachen" Multiplikationsalgorithmus.

Es tritt hier aber ein zusätzliches Problem auf: Alle Divisionsalgorithmen arbeiten nur dann sinnvoll, wenn der Divisor nicht "zu klein" ist, etwa falls, wie bei uns, die Zahl der Stellen des Dividenden gleich der Summe der Stellenzahlen von Divisor und Quotient ist (einmal ohne Berücksichtigung von Vorzeichenstellen) - wenn also der Divisor nicht mehr führende Nullen hat als der Dividend. Wir bekommen sonst Quotientenüberlauf und zwar lauter Nullen oder lauter Einsen, was ja andererseits Codierungen von "null" oder "minus eins" sind. Deshalb müssen Komponentenpaare, für die Überlauf auftreten wird, vorher erkannt werden. Am besten führt man für solche Komponenten die Division gar nicht erst aus, um die Operanden nicht zu zerstören. Ein einfacher Algorithmus besteht darin, den ersten Subtraktions-/Additions-Schritt der eigentlichen Division zweimal auszuführen, um diejenigen Komponenten zu finden, bei denen dann immer noch kein Vorzeichenwechsel des Dividenden-Restes aufgetreten ist. Praktisch: Man führt den ersten Divisionsschritt einmal aus, um eine Stelle nach "oben" verschoben für den Dividenden (entspricht Faktor 2), schreibt aber die Ergebnisbits dieser Operation nicht auf; der Dividend wird dabei um eine Wiederholung des Vorzeichenbits nach "oben" verlängert gedacht. Laufzeitabschätzung: $O(\frac{n}{r} \cdot \ell)$ wie bei einer Addition/Subtraktion.

3.4.1.5 Quadratwurzel

Eine solche Operation steht in konventioneller Hardware nicht zur Verfügung und muß durch "numerische Iteration" berechnet werden. Dagegen ist sie verhältnismäßig einfach durch Pipeline-Arrays implementierbar und kann auch als Vertikaloperation implementiert werden.

Die Vertikaloperation ist für $p=1$ (d.h. $n=r$) aufgeschrieben. Der entsprechende Faktor ist aber in den Laufzeitabschätzungen allgemein berücksichtigt.

Der Algorithmus arbeitet ohne "Rekonstruktion des Restes": Es wird, wenn die Stellen $s_0, s_1, \dots, s_i$ der Quadratwurzel bereits berechnet sind, zur Berechnung der Stelle s_{i+1} vom Rest des Radikanden die Zahl $00\dots 0s_0s_1\dots s_i01$ (mit $i+1$ führenden Nullen) abgezogen; wenn der Rest positiv ist, ist $s_{i+1}=1$, es wird entsprechend wieder subtrahiert; wenn der Rest negativ ist, ist $s_{i+1}=0$; im nächsten Schritt wird die Zahl $00\dots 0s_0s_1\dots s_{i+1}11$ (mit $i+2$ führenden Nullen) addiert.

par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

$loadmask\{m_j\};$

$d_j := 1;$

$x_j := \neg d_j;$

"vorbesetzen Operationenmarkierung und Wert der zweituntersten Stelle bei Subtraktion/Addition"

sequ $i = 0, 1, \dots, l-2$

"Schleife über die Zahlendarstellung des Ergebnisses"

$loadregister\{d_j, \text{null}\};$ *)

"vorbesetzen Hilfsregister des Zusatz-Moduls"

$read\ a_{2i+1,j};$

$y_j := 1;$

$y_j := addsub\{a_{2i+1,j}, y_j\};$ *)

$a_{2i+1} := mask\{a_{2i+1,j}, y_j\};$ *)

$write\ a_{2i+1,j};$

"subtrahieren/addieren eine 1"

*) genaue Beschreibung in 3.5.2.1

```

read  $a_{2i,j}$ ;
 $y_j := \text{addsub } (a_{2i,j}, x_j);$  *)
 $a_{2i,j} := \text{mask } (a_{2i,j}, y_j);$  *)
write  $a_{2i,j}$ ;
"subtrahieren/addieren eine 0/1"

sequ  $k = i, i-1, \dots, 0$ 
"Schleife über die schon berechneten Stellen des Ergebnisses"

read  $a_{i+1-k,j}$ ;
read  $s_{i,j}$ ;
 $y_j := \text{addsub } (a_{i+1-k,j}, s_{i,j});$  *)
 $a_{i+1-k,j} := \text{mask } (a_{i+1-k,j}, y_j);$  *)
write  $a_{i+1-k,j}$ ;
"subtrahieren/addieren Stelle des Zwischenergebnisses"

end sequ;

sequ  $h = i, i-1, \dots, 0$ 
"Schleife über die führenden Nullen"

read  $a_{i,j}$ ;
sim  $y_j := \text{addsub } (a_{i,j}, \text{null})$  *)
with  $c := \dots$ 
end sim;

 $a_{i,j} := \text{mask } (a_{i,j}, \text{null})$  *)
write  $a_{i,j}$ ;

if  $c \text{ equal } 0$ 
"Abfrage, ob kein Übertrag mehr"

then read  $a_{0,j}$ ;
exit

else

end if

end sequ;

```

*) genaue Beschreibung in 3.5.2.1

```

sim read  $s_{i+1,j}$ 
with  $x_j := a_{\emptyset,j} \wedge m_j$ ;
      "ausblenden Vorzeichen des Radikandenrestes"
       $y_j := x_j \oplus m_j$ 
      "Komplement des Vorzeichens ist Ergebnisbit"

end sim;

 $s_{i+1,j} := \text{mask}(s_{i+1,j}, y_j);$           *)
sim write  $s_{i+1,j}$ 
with  $d_j := y_j$ 
      "von Ergebnisbit hängt ab, ob subtrahiert oder addiert wird"
end sim

end sequ

end par

```

Auch hier läßt sich die kombinierte Addition/Subtraktion ausnutzen. Wir haben insgesamt

$$\sum_{i=\emptyset}^{\ell-1} 2(i+1) = (\ell+1)(\ell+2)$$

Einzelsschritte (Additions/Subtraktionsschritte). Die Laufzeit insgesamt ist also $O(\frac{n}{r} \cdot \ell^2)$.

3.4.1.6 Komponentensumme

Zur Unterstützung der Vektorrechnung im mathematischen Sinn ist dieser Befehl in die Liste aufgenommen worden. Die horizontale Blockoperation stellt keine Probleme und ist deshalb hier nicht weiter interessant. Die *Vertikaloperation* ist zunächst für *Standard-Hardware* beschrieben. Das Ergebnis soll eine horizontal gespeicherte binäre Festkommazahl sein. Für ein System ohne Orthogonalspeicher muß außerdem die Möglichkeit bestehen, das Ergebnis als Komponente eines Vektors in Vertikalardarstellung zu erzeugen, d.h. bitweise in einer markierten Spalte abzuspeichern.

*) genaue Beschreibung in 3.5.2.1

$v := \emptyset$

sequ $h = \emptyset, 1, \dots, q-1$

"Schleife über die Teilworte des Ergebnisses"

$\overline{s}(h) := \underline{null};$

$\overline{x} := \underline{null};$

"vorbesetzen Teilwort von Summe und Hilfsvariable"

sequ $i = r-1, r-2, \dots, \emptyset$

"Schleife über die Bits eines Teilwortes"

sequ $k = \emptyset, 1, \dots, p-1$

"Schleife über die Teilvektoren mit je r Komponenten"

par $j = \emptyset, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. des Rechenwerkes"

$\text{read } a_{i,kr+j};$

$y_j := a_{i,kr+j} \wedge m_{kr+j}$

"lesen Operanden und ausblenden ausgewählte Komponenten"

end par;

sequ $j = \emptyset, 1, \dots, r-1$

"Schleife über die Bits eines Registers zum Feststellen der Werte der Komponenten-Bits des Operanden"

if $y_j \text{ equal } 1$

"Abfrage, ob Bit den Wert 1 hat"

then $\overline{x} := \text{add } (\overline{x}, \overline{1})$

"konventionelle Festkomma-Addition einer Eins auf Summe"

else

end if

end sequ;

$(\overline{x}, \overline{s}) := \text{shl } (\overline{x}, \overline{s})$

"shiften nach rechts logisch um eine Stelle das Doppelwort $\overline{x}$ verkettet mit $\overline{s}$ "

end sequ

end sequ;

write $\overline{s}(h)$

"abspeichern Teilwort der Summe"

end sequ;

if $\bar{x} \text{ unequal null}$

"Abfrage, ob Teile der Summe nicht abspeicherbar"

then $o := 1$

"setzen Overflow-Flag"

else

end if

Dieser Algorithmus arbeitet voll-sequentiell auf dem gesamten Datenbestand (n Bits). Der normale Parallelismus des Rechenwerks läßt sich nicht ausnutzen. Der kritische Punkt ist das Auszählen der Bits mit dem Wert 1 in einem Register. Mit geeigneter Zusatz-Hardware - dem "Quersummen-Modul", vgl. 3.5.2.2 - läßt sich der Algorithmus wesentlich beschleunigen. Wir ersetzen die Anweisungsfolge in der dritten Schleife (von außen gezählt) durch:

par $j = 0, 1, \dots, n-1$

"normale Parallelarbeit des Speichers bzw. des Rechenwerkes"

read $a_{i,kr+j}$;

"lesen Operand"

$\bar{y} := \text{sum } (a_{i,kr+j}, m_{kr+j});$ *)

$\bar{x} := \text{add } (\bar{x}, \bar{y})$

end par;

$(\bar{x}, \bar{s}) := \text{shift} (\bar{x}, \bar{s})$

"shiften nach rechts logisch um eine Stelle das Doppelwort $\bar{x}$ verkettet mit $\bar{s}$ "

3.4.1.6 Komponentenweiser Vergleich

Die horizontale Blockoperation liefert gegenüber der Addition/Subtraktion nichts wesentlich Neues: Man subtrahiert und notiert, ob die Ergebnisse größer, kleiner, gleich null sind. Wenn man dabei mit dem höchstwertigen Teilwort beginnt, kann man außer "Gleichheit" diese Informationen schon mit einem Teil der Einzelzahlen gewinnen.

*) genaue Beschreibung in 3.5.2.2

Bei der Vertikaloperation wird, beginnend mit dem höchstwertigen Bit, Bit für Bit verglichen: Bei Nichtmehrerübereinstimmen wird festgestellt, ob der erste Operand größer oder kleiner als der zweite ist. Das Ergebnis seien drei Masken $\overline{m(=)}$, $\overline{m(>)}$, $\overline{m(<)}$. Mit Hilfe einer Operationenmarkierung $\overline{d}$ kann eine *gemeinsame* Implementierung für arithmetische und lexikalische Vergleiche durchgeführt werden: $d=1$ für arithmetische Vergleiche, $d=0$ für lexikalische Vergleiche; das heißt: mit und ohne Berücksichtigung der Sonderrolle des binären Vorzeichens. In der Implementierungsskizze ist die äußerste Schleife weggelassen.

par $j = 0, 1, \dots, n-1$

$m(=)_j := m_j;$

$m(>)_j := 0;$

$m(<)_j := 0;$

"vorbesetzen die drei Ergebnismasken"

sequ $i = 0, 1, \dots, l-1$

"Schleife über die Bits der Zahlendarstellung"

read $a_{i,j};$

read $b_{i,j};$

"lesen Vergleichsoperanden"

$x_j := \text{equivalence}(a_{i,j}, b_{i,j});$ *)

"feststellen Übereinstimmungen"

$y_j := m(=)_j \oplus x_j;$

$m(=)_j := x_j;$

$a_{i,j} := a_{i,j} \oplus d_j;$

$z_j := a_{i,j} \wedge y_j;$

$m(>)_j := m(>)_j \vee z_j;$

$z_j := y_j \oplus z_j;$

$m(<)_j := m(<)_j \vee z_j;$

"bei Ungleichheit feststellen, welcher Operand größer ist"

*) genaue Beschreibung in 3.5.2.2

```

if m(=) equal null
    "Abfrage, ob keine Übereinstimmungen mehr"
then exit
else
end if;

 $d_j := \emptyset$ 
end sequ
end par

```

Die gemeinsame Implementierung für logische *und* arithmetische Vergleiche kostet in der Schleife zwei zusätzliche Befehle. Laufzeit $O(\frac{n}{r} \cdot \ell)$.

3.4.1.8 Arithmetisches Maximum/Minimum

Bei der horizontalen Komplexoperation erhält man sofort den Wert des Maximums, aber, ohne größeren Aufwand, den Index nur des ersten Auftretens dieses Wertes. Bei der Vertikaloperation erhält man die Markierung aller Komponenten mit dem Maximum, aber nicht den Wert.

Die horizontale Blockoperation ist in Anhang A.3 angegeben. Dort wird ein Algorithmus benutzt, der mit mehrfachgenauen Subtraktionen arbeitet und dabei alle Bits des Zahlenbestandes genau einmal verarbeitet. Ein schnellerer Algorithmus würde ähnlich vorgehen wie der vertikale: Man vergleiche zunächst nur das oberste Teilwort mit dem des bisherigen Maximums; erst wenn sich damit keine Entscheidung treffen läßt, vergleiche man das nächste usw. Dabei müssen evtl. etwas kompliziertere Sequenzen statt einfacher Subtraktionen benutzt werden (damit nicht jeweils die obersten Bits von Teilworten als Vorzeichenbits interpretiert werden), aber man braucht insgesamt im allgemeinen weniger Operationen.

Mit einer Maske $\bar{d}$ zur Operationencodierung läßt sich für die Vertikalverarbeitung eine *gemeinsame* Implementierung von Maximum- und Minimum-Suche angeben: $d_j = \emptyset$ bedeutet "Maximumsuche", $d_j = 1$ bedeutet Minimumsuche. Mit einer zweiten solchen

Maske $\overline{f}$ lassen sich aber auch die Implementierungen für *arithmetische* und *lexikalische Maximum-/Minimum-Suche* zusammenfassen: $f_j = 0$ bedeutet "ohne Sonderbehandlung des höchstwertigen Bit"; $f_j = 1$ bedeutet "das höchstwertige Bit ist als Vorzeichenbit zu behandeln".

sequ $k = 0, 1, \dots, p-1$

"Schleife über die Teilvektoren mit je r Komponenten"

par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerks"

$$z_{kr+j} := d_{kr+j} \oplus f_{kr+j}$$

"komplementieren Operationenmarkierung d zur Sonderbehandlung der Vorzeichenbits, wenn arithmetische Vergleiche"

end par

end sequ;

sequ $i = 0, 1, 2, \dots, l-1$

"Schleife über die Bits einer Zahlendarstellung"

par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerks"

$$x_j := 0$$

end par;

sequ $k = 0, 1, \dots, p-1$

"Schleife über die Teilvektoren mit je r Komponenten"

if $\bar{m}^{(k)}$ unequal null

then par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerks"

read $a_{i,kr+j};$

$$a_{i,kr+j} := a_{i,kr+j} \oplus z_{kr+j};$$

$$y_{kr+j} := a_{i,kr+j} \wedge m_{kr+j}$$

"ausblenden Bits der noch interessierenden Komponenten"

end par;

sequ $j = 0, 1, \dots, n-1$

"Schleife über die Bits eines Registers"

if y_{kr+j} equal 1

"Abfrage, ob Bit Wert 1 hat"

then $\bar{x} = \text{add}(\bar{x}, 1)$

"konventionelle Addition zum Zählen der Einsen"

else

end if

end sequ

else

end if

end sequ;

if x unequal null

"Abfrage, ob weitere Unterscheidungen zwischen den noch interessierenden Komponenten möglich geworden sind"

then sequ $k = 0, 1, \dots, p-1$

"Schleife über die Teilvektoren mit je r Komponenten"

par $j = 0, 1, \dots, n-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

$m_{kr+j} := y_{kr+j}$

"neue Maske zur Auswahl der immer noch interessierenden Komponenten"

end par

end sequ;

if $\bar{x}$ equal 1

"Abfrage, ob genau ein Maximum diskriminiert wurde"

then exit

"Abbrechen des Suchens"

else

end if

else

end if;

sequ $k = 0, 1, \dots, p-1$

"Schleife über die Teilvektoren mit je r Komponenten"

par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

$\bar{z}_{kr+j} := d_{kr+j}$

"übernehmen ursprüngliche Operationencodierung"

end par

end sequ

end sequ;

sequ $k = 0, 1, \dots, p-1$

"Schleife über die Teilvektoren mit je r Komponenten"

par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

write m_{kr+j}

end par

end sequ

Wie die Komponentensumme ist dieser Algorithmus voll-sequentiell für die $n \cdot \ell$ Bits des Datenbestandes und zwar durch den Teil, in dem die Bits mit dem Wert 1 in der Maske ausgezählt werden. Bei Vertikalarbeit auf der Standard-Hardware ist es deshalb besser, auf den vorzeitigen Abbruch bei eindeutig festgestelltem Extremum zu verzichten, stattdessen die Daten in ihrer gesamten Länge ℓ zu durchsuchen und dabei den Wert des Extremums festzustellen. Ein solcher Algorithmus hat Laufzeit von $O(\frac{n}{r} \cdot \ell)$.

Mit Zusatz-Hardware kann aber der ursprüngliche Algorithmus beschleunigt werden. Wir benutzen auch hier den "Quersummen-Modul" (vgl. 3.5.2.1), um die Summe der Bits eines Registers zu bilden. Für den Spezialfall $p=1$ kann eine zusätzliche konkrete Beschleunigung erreicht werden, wenn die Operation "sum" ein Markierungsbit erzeugt für die Feststellung "Summe hat den Wert 1". Mit Zusatz-Hardware haben wir eine Laufzeit von $O(\frac{n}{r} \cdot \ell)$, aber im allgemeinen braucht nicht der ganze Datenbestand abgearbeitet zu werden: Bei gleichverteilten Daten braucht man, grob geschätzt, dann im allgemeinen statt ℓ Schritten $\log_2 r$ Schritte.

3.4.1.9 Schwellwertvergleich

Für die horizontale Blockoperation ergibt sich nichts Neues gegenüber dem komponentenweisen Vergleich. Für die Vertikaloperation kann man den Schwellwert wie einen "Skalar" benutzen, der "dynamisch" zum Vektor mit lauter gleichen Komponenten vervielfacht wird - vgl. 3.4.1.7.

3.4.1.A Zählen addierend/subtrahierend

Die horizontale Blockoperation wird ähnlich der allgemeinen Addition/Subtraktion implementiert; wieviele Teilworte einer Zahl dabei bearbeitet werden müssen, hängt davon ab, ob noch ein Übertrag aus den unteren Worten zu verarbeiten ist.

Die Vertikaloperation arbeitet mit speziellen Additions-/Subtraktionsschritten: Komplementierung der Bits der Operanden, beginnend mit dem niedrigstwertigen bis einschließlich der ersten Null/Eins. Wir haben wieder eine *gemeinsame* Implementierung der beiden Aufgaben, mit einer "Operationenmarkierung" $\vec{d}$: $d_j = 0$ bedeutet Addition; $d_j = 1$ bedeutet Subtraktion.

In der Beschreibung ist die äußere Schleife über die Teilvektoren mit je r Komponenten als unwesentlich weggelassen; in der Laufzeitabschätzung ist sie aber berücksichtigt.

par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

$x_j := m_j;$

sequ $i = l-1, l-2, \dots, 0$

"Schleife über die Bits der Zahlendarstellung"

if $\bar{x}$ equal null

"Abfrage, ob $\bar{x}$ als Festkommazahl null geworden ist"

then exit

else read $a_{i,j};$

$a_{i,j} := a_{i,j} \oplus x_j;$

"komplementieren Bits der Operanden"

sim write $a_{i,j}$

with $y_j := a_{i,j} \oplus d_j;$

$y_j := y_j \wedge m_j;$

$x_j := x_j \oplus y_j$

"reduzieren $\bar{x}$ um diejenigen Bits, bei denen letzter Schritt der Komplementierung ausgeführt wurde, d.h. von unten her erste 0 bzw. 1 "

end sim

end i;

end sequ

end par

Durch Zusatz-Hardware, etwa den "Vertikal-Modul" (vgl. 3.5.2.1), können die vier Verknüpfungsoperationen zusammengezogen werden zu einer. Diese Operation muß ein Markierungsbit liefern für "kein Übertrag erzeugt", um nur die minimale Anzahl von Speicherworten (Bit-Slices) verarbeiten zu müssen. Laufzeit: $O(\frac{n}{r} \cdot \ell)$.

3.4.2 Assoziative Verarbeitung

"Assoziativ" soll speziell bei den Suchoperationen bedeuten, daß, aus der Sicht des Benutzers, in einem ganzen Block von Daten "auf einmal" gesucht wird. Im allgemeinen wird dabei nur in vorher als relevant markierten Teilen der Eintragungen gesucht. Analog dazu: mehrfache Eintragungen.

Das Prinzip der Vertikalverarbeitung besteht darin, die Datenworte (im Sinne ihrer Interpretation) wort-parallel bit-seriell zu verarbeiten. Die interessierenden Eintragungen werden durch eine Maske ausgewählt; die Antworten auf die Suchoperationen sind ebenfalls Markierungen durch Masken. Die relevanten Teile innerhalb der Datenworte werden durch eine weitere Maske ("Relevanzmaske") markiert. Die Auswahlmaske entspricht den "Selektoren" assoziativer Speicher, die Relevanzmaske deren Maskenregister.

Zu diskutieren bleibt, wie Kriterien und Relevanzmasken vorgegeben sind - konventionell gespeichert oder als Komponenten von Vektoren in Vertikaldarstellung. Mit einem Orthogonalspeicher wäre das aber kein wesentlicher Unterschied mehr.

Die Befehlsliste:

1. Suchen unter ausgewählten Datenworten nach Übereinstimmung mit Suchkriterium in relevanten Teilen - "Exact Match"
2. Suchen unter ausgewählten Datenworten nach bestmöglicher Übereinstimmung mit Suchkriterium in relevanten Teilen - "Best Match" (minimaler Hamming-Abstand)
3. Eintragen in ausgewählten Datenworten relevante Teile eines vorgegebenen Datenwortes
4. Suche nach dem lexikalischen Maximum unter ausgewählten Datenworten
5. Suche nach dem lexikalischen Minimum
6. Schwellwertsuche - Markierung, welche Datenworte lexikalisch kleiner, gleich, größer einem vorgegebenen Kriterium sind

Konvention (Vollständige Liste siehe Seiten 166 und 167):

$a_{i,j}$	sind Bits des Operandenfeldes
b_j	sind Bits des Kriteriums
m_i bzw. m_j	sind Bits der Selektionsmaske
e_j	sind Bits der Relevanzmaske
$x_{i,j}, y_{i,j}$ usw.	sind Bits von Zwischenmasken

3.4.2.1 "Exact Match"

Die horizontale Blockoperation kann aus konventionellen Subtraktionen aufgebaut werden; Operanden und Suchkriterium müssen vorher jeweils mit der Relevanzmaske ausgeblendet werden. Laufzeitabschätzungen: $O(n \cdot \frac{L}{r})$.

Die *Vertikaloperation* hat demgegenüber den Vorteil, daß sie bei Erkennen von Mißerfolg abgebrochen werden kann. Bei der Implementierungsskizze ist die äußere Schleife (Einteilung der Daten in Blöcke von r Eintragungen) wegge-

lassen; in den Abschätzungen ist sie berücksichtigt. Es wird Zusatz-Hardware ("Vertikal-Modul" - vgl. 3.5.2.1) vorausgesetzt.

par $j = \emptyset, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerks"

sequ $k = \emptyset, 1, \dots, q-1$

"Schleife über die Teilwörter von Kriterium und Relevanzmaske"

sequ $h = \emptyset, 1, \dots, r-1$

"Schleife über die Bits in den Teilwörtern"

read $a_{kr+h,j};$

"lesen Bits der Operanden"

if $e_h \text{ unequal } \emptyset$

"Abfrage, ob Bit nicht relevant"

then $m_j := \text{compare } (a_{kr+h,j}, b_h) \quad *)$

"Maske reduzieren auf diejenigen Positionen, wo Operanden-bit denselben Wert hat wie Kriteriumsbit"

else

end if;

if $\overline{m} \text{ equal null}$

"Abfrage, ob Auswahlmaske für den nächsten Schritt null geworden ist"

then exit

else

end if

end sequ

end sequ;

write m_j ;

"schreiben Ergebnismaske"

end par

*) genaue Beschreibung in 3.5.2.1

Für viele Anwendungen (vgl. auch Kapitel 4) braucht man ein Suchen nach mehreren Kriterien unter derselben Relevanzmaske. Hardware-realisierte assoziative Systeme erlauben eine solche Operation nicht, weil dort i.a. nur ein Ein-/Ausgaberegister bzw. Datenregister zur Verfügung steht. In einem mikroprogrammierten Konzept läßt sich eine solche Operation prinzipiell implementieren, indem man die Schleife über alle Kriterien (Schlüssel) sozusagen zur innersten Schleife des Mikroprogrammes macht. Schwierig wird aber die Speicherung der Schlüssel in einer Form, die möglichst wenig Speicherzugriffe erfordert, und vor allem das Führen vieler Ergebnismasken, nämlich einer Maske pro Schlüssel, die im allgemeinen Fall nicht mehr in Registern gehalten werden können.

Mit einem Hardware-Skalierbefehl, etwa im "Quersummen-Modul" (vgl. 3.5.2.2) kann die Abfrage der Relevanzmaske beschleunigt werden: Man kann Folgen von Nullen überspringen durch entsprechende Auswertung des Skalierwertes bei der Adressenrechnung.

3.4.2.2 "Best Match"

Unter den vielen möglichen Definitionen von "Best Match" - etwa: numerisch kleinster Abstand; Übereinstimmung in der größten Anzahl führender Bits usw. - soll hier diejenige durch den "geringsten Hamming-Abstand" benutzt werden. Das Ergebnis der Operation soll der Wert dieses Abstandes sein sowie eine Markierung der Dateneintragung bzw. Dateneintragen mit minimalem Abstand vom Kriterium.

Sowohl bei der horizontalen Blockoperation wie bei der Vertikaloperation müssen Hilfsvektoren mit der Angabe der Übereinstimmungen angelegt werden; gesucht wird dann in diesem Hilfsvektor das Maximum, als Zeiger und als Wert. Dabei liefert wieder die horizontale Blockoperation sofort den Wert und einen Zeiger, d.h. ein möglichst gut übereinstimmendes Datenwort. Die Vertikaloperation liefert alle Zeiger (als Markierungsmaske) und den Wert der Abweichung. Bei der horizontalen Blockoperation werden die Komponenten des Hilfsvektors nicht

gespeichert, sondern nur mit dem bisherigen Maximum verglichen. Bei der Vertikaloperation wird ein Vektor in Vertikaldarstellung aufgebaut. Der Bereich dieses Hilfsvektors wird als gelöscht vorausgesetzt. Der vertikale Hilfsvektor heie u . Die Komponenten des Hilfsvektors brauchen eine Zahlendarstellung mit maximal $\lceil \log_2 \ell \rceil$ Bits. Fr die praktische Anwendung soll angenommen werden, da ℓ als konventionelle Festkommazahl dargestellt werden kann, also $\lceil \log_2 \ell \rceil \leq r$.

Die horizontale Blockoperation findet sich in Anhang A. Sie ist ein Beispiel dar, da auch horizontale Operationen gelegentlich von unkonventionellen Mikrobefehlen der Zusatz-Hardware profitieren knnen: In diesem Falle kann der "Quersummen-Modul" bentzt werden, vgl. 3.5.2.2. Der folgende Algorithmus fr die *Vertikalverarbeitung* gebraucht dagegen den "Vertikal-Modul".

sequ $k = 0, 1, \dots, p-1$

"Schleife ber die Teilvektoren mit je r Komponenten"

par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerks"

sequ $i = 0, 1, \dots, \ell-1$

"Schleife ber alle Bits einer Zahlendarstellung"

if e_i equal 1

"Abfrage, ob relevante Stelle"

then read $a_{i,kr+j}$;

$x_j := \text{equivalence}(a_{i,kr+j}, b_i);$ *)

$x_j := x_j \wedge m_{kr+j}$

"feststellen bereinstimmende Bits in markierten Spalten"

else

end if;

sequ $h = \lceil \log_2 \ell \rceil - 1, \lceil \log_2 \ell \rceil - 2, \dots, 0$

"Schleife ber die Bits der Zahlendarstellung des Hilfsvektors"

if $\bar{x}$ unequal null

"Abfrage, ob noch Werte zu verknpfen"

*) genaue Beschreibung in 3.5.2.1

then read $u_{h,kr+j}$;

$u_{h,kr+j} := u_{h,kr+j} \oplus x_j$;

$y_j := u_{h,kr+j} \wedge x_j$;

$x_j := y_j \oplus x_j$

"spezielle Additionsverknüpfung zum Zählen: kompletieren Bits einer Zahlendarstellung bis einschließlich erster 0"

else

end if

end sequ

end sequ

end par

end sequ

Bemerkungen: Auf dem Hilfsvektor u muß nun, wie in 3.4.1.8 beschrieben, das arithmetische Maximum gesucht werden.

Laufzeitabschätzungen: $O(\frac{n}{r} \cdot \ell \cdot \log_2 \ell + \frac{n}{r} \cdot \log_2 \ell)$ also $O(\frac{n}{r} \cdot \ell \cdot \log_2 \ell)$

Es ergibt sich also, da $\log_2 \ell \leq r$ vorausgesetzt war, eine etwas günstigere Abschätzung als für die horizontale Blockoperation mit Standard-Hardware; vor allem ist zu berücksichtigen, daß Zähloperation und die Maximumsuche nur in ungünstigen Fällen über die ganze Länge der Zahlendarstellung des Hilfsvektors ausgeführt werden müssen.

Auch bei "Best Match" ist ein Skalierbefehl gut zu gebrauchen.

3.4.2.3 Einschreiben

Diese Operation läßt sich sowohl als horizontale Blockoperation wie als Vertikaloperation einfach aufbauen aus logischen Befehlen zum Maskieren und Zusammensetzen.

Laufzeitabschätzungen für beide Implementierungen: $O(\frac{n \cdot \ell}{r})$

3.4.2.4 Lexikalisches Maximum/Minimum

Für die horizontale Blockoperation bedeutet es einigen Aufwand, die Vorzeichenbits vorab auszuwerten. Die Blockoperation soll hier nicht im Detail ausgeführt werden.- Für die Vertikaloperation ist in 3.4.1.8 ein *allgemeiner* Algorithmus angegeben, der die hier gestellte Aufgabe ebenfalls erfüllt.

3.4.2.5 Schwellwertvergleich

Gegenüber dem arithmetischen Schwellwertvergleich ergibt sich auch hier, daß das höchstwertige Bit nicht als Vorzeichenbit behandelt werden darf. Das bedeutet: Bei der horizontalen Blockoperation müssen die höchstwertigen Bits zunächst gesondert abgefragt werden, ehe Subtraktionen benutzt werden können. Die Vertikaloperation ist bereits in 3.4.1.9 beschrieben.

3.4.3 Muster-Verarbeitung

Hier sollen kurz einige mögliche Hilfsmittel zur Bearbeitung zweidimensionaler binärer Muster skizziert werden.

Als einfachste Hilfsmittel haben wir bitweise logische Verknüpfungen für alle Bits des Feldes. Die Implementierungen durch horizontale Blockoperationen oder durch Vertikaloperationen unterscheiden sich nur durch eine Vertauschung von Laufschleifen. Ob wir dabei Auswahlmasken benutzen wollen, hängt von den vorgesehenen Anwendungen ab. Unter Umständen ist es sinnvoll, mit zwei solchen Masken zu arbeiten, für die Zeilen und die Spalten der Bit-Matrix, etwa für Fenster.

Mit Hilfe der Hardware-Zusätze für die Datenkonversion - "Drehscheibe", Orthogonalspeicher o.ä. - können wir Spiegelungen von $r \times r$ -Bit-Matrizen an ihren Diagonalen und Drehungen um $+90^\circ$ und -90° unmittelbar implementieren. Die entsprechenden Operationen für größere Bit-Matrizen können ebenfalls verhältnismäßig einfach aufgebaut werden.

Shift-Operationen für ganze Bit-Matrizen sind ebenfalls sehr einfach implementierbar durch Hardware-Shifts und Umspeicherungen (bzw. Versetzen von Adreß-Zeigern). Aus diesen Operationen lassen sich aufbauen: Schrumpfen eines Musters, Aufblähen, Rändern /24/.

Ein anderer Typ von Operationen, zu dem übrigens bereits die zuletzt zitierten gehören, arbeitet auf Nachbarschaften von Punkten: Der Funktionswert für einen Punkt hängt ab vom Wert dieses Punktes und den Werten der vier (bzw. acht) unmittelbaren Nachbarpunkte. Hier haben wir, als mögliche Implementierung, etwa eine Schleife über alle Zeilen der Bit-Matrix; zur Berechnung der neuen Werte in Zeile i werden benutzt: Die Zeilen $i-1$, $i+1$ und zwar sowohl zu "spalten-gleichen" logischen Verknüpfungen wie auch zu Verknüpfungen mit vorhergehenden Shift um eine Stelle nach links oder nach rechts. Es bleibt zu diskutieren, ob Probleme mit solchen Operationen häufig genug sind, damit es sich lohnt, einen Spezial-Modul dafür zu entwerfen. Beispiele für solche Probleme bzw. Algorithmen zu deren Lösung finden sich in /28/ und /49/.

Außerdem gibt es Probleme, bei denen "gerechnet" werden muß, z.B. das Auffinden des Schwerpunktes von allen "schwarzen Punkten". Algorithmen dafür lassen sich etwa mit Hilfsmitteln, die für die Vektor-Arithmetik bereitgestellt sind (wie der Summationsbefehl und die Drehscheibe) vergleichsweise gut vertikal implementieren. Dies scheint übrigens, außer den Dreh- und Spiegelbefehlen, der einzige Typ von Operationen zu sein, wo die Vertikalverarbeitung qualitativ etwas anderes leistet als die Vertauschung von Laufindices.

Auf die Zusammenstellung einer Befehlsliste und die Beschreibung von Implementierungsansätzen wird hier verzichtet.

3.4.4 Hilfsoperationen

Hier handelt es sich teilweise um Operationen, die unabhängig von der Implementierung zur Ergänzung der Blockoperationen nötig sind wie etwa Vorbesetzen eines Blockes mit null usw. Teilweise handelt es sich um Spezialbefehle für die Vertikalverarbeitung. Dabei zeigt sich, daß man für ein System mit "Drehscheibe" Befehle braucht, die beim Orthogonalspeicher nicht nötig sind: Konversion von (konventioneller) Horizontal- in Vertikal-Darstellung und umgekehrt; Einschreiben

einer horizontal vorgegebenen Information in ausgewählte Spalten eines Vertikalfeldes; Herauslesen einer markierten Spalte eines Vertikalfeldes in Horizontal-darstellung.

Speziell für die Vertikalverarbeitung sinnvoll ist ein Befehl zum "Zusammenpacken" markierter Spalten aus mehreren Teilblöcken mit je r Spalten, zum Beispiel um bei assoziativer Datenverarbeitung die interessierenden Daten zur Weiterverarbeitung so stehen zu haben, daß man die Parallelarbeit wieder möglichst gut ausnutzen kann.

Außerdem braucht man, als Unterstützung für die höhere Programmiersprache, in der das Konzept zur Verfügung stehen soll, Befehle zum Ändern von Auswahl- und Relevanzmasken, Parameterlisten, Deskriptoren usw.

Auch hier soll auf eine detaillierte Befehlsliste mit Implementierungsskizzen verzichtet werden.

3.5 Auswertung

In diesem Abschnitt sollen die formalen Beschreibungen ausgewertet werden. Davon zu erwarten ist eine Bewertung der verschiedenen Implementierungsansätze, eventuell auch der benutzten Algorithmen und der zugrundeliegenden Hardware.

Beim Vergleich des mikroprogrammierten Parallel- und Pseudoassoziativ-Konzeptes mit der Spezial-Hardware (assoziatives System, arithmetische Pipeline) zeigt sich, sowohl für die horizontalen Blockoperationen wie für die Vertikaloperationen, daß die mikroprogrammierten Implementierungen - was ja zu erwarten ist - wesentlich höhere Flexibilität erlauben, etwa in der Länge ℓ der Daten-darstellung. Diese Flexibilität auf Benutzer-Ebene muß zwar auf Mikroprogramm-Ebene durch Komplexität bei der Implementierung erkaufte werden, ist aber prinzipiell machbar. Dagegen kann etwa in einem assoziativen System ein längeres Datenformat nur verarbeitet werden, wenn es Möglichkeiten gibt, den "Kontext" eines Speicherwortes aufzufinden, etwa die Selektoren um eine oder mehrere Stellen nach oben oder unten zu verschieben, wie es zum Beispiel in /50/ beschrieben ist. Solche Möglichkeiten seien für die Abschätzungen vorausgesetzt.

Auch die arithmetischen Pipeline-Arrays sind, wie konventionelle Rechenwerke, auf eine Datenlänge festgelegt. Es besteht natürlich auch hier die Möglichkeit, mehrfach genaue Arithmetik per Programm aufzubauen. Die Zeitabschätzungen für Pipelines (vgl. 3.1.2) sind nicht in die Tabelle aufgenommen.

3.5.1 Laufzeitvergleiche für verschiedene Maschinenkonzepte und Implementierungen

In Tabelle 3.1 sind Vergleiche der horizontalen Blockoperationen und der Vertikaloperationen zusammengestellt, sowie für assoziative Systeme. Zu den einzelnen Spalten:

1. Operationen sind zusammengefaßt, wo sich horizontale Blockoperationen jeweils nur um einzelne Mikrobefehle unterscheiden (Addition/Subtraktion) und die entsprechenden Vertikaloperationen eine gemeinsame Implementierung haben.
2. Implementierung: Arbeitsweise und Hardware. Es bedeuten: h horizontal, v vertikal; s konventionelle Standard-Hardware, z Zusatz-Hardware; a assoziatives System.
- 3./4. Die Abhängigkeiten von den Dimensionsparametern n und ℓ , der Speichervortlänge bzw. Rechenwerkbreite r und den Einsen in der Auswahlmaske m und der Relevanzmaske e (geschrieben als $|m|$, $|e|$) sind durch Auszählen der wesentlichen Laufschleifen (ohne Vor- und Nachbereitung) abgeschätzt.
5. Die "absoluten" Laufzeiten einer "innersten" Schleife sind angegeben als Vielfache der Operationszeit einer logischen Verknüpfung und abgeschätzt mit folgender Konvention: Logische Verknüpfungen, Schiebebefehle, Sprünge, Addition, Subtraktion brauchen eine Zeiteinheit (- hier geht wesentlich die Hardware-Philosophie heutiger Minirechner ein -); eine Schleifenabfrage braucht zwei Zeiteinheiten; ein Speicherzugriff braucht zwei Zeiteinheiten; Multiplikation, Division brauchen 11 Zeiteinheiten (ohne Berücksichtigung der impliziten Abhängigkeit von r). Diese Werte sind grobe Verallgemeinerungen der konkreten Zahlen für die Interdata 85. In die Abschätzungen für die "absoluten" Laufzeiten aufgenommen sind konstante Faktoren, die in der Schreibweise $O(n, \ell, r, |m|, |e|)$ nicht explizit auftreten. Für die Spezial-Hardware sind solche Abschätzungen nicht durchgeführt.

Tabelle 3.1: Vergleich der horizontalen Blockoperationen und der Vertikaloperationen

Operation	horizontal/ vertikal, Standard-/ Zusatz- Hardware, assoziativ	Laufzeit abhängig von n, ℓ, r	Laufzeit abhängig von n, ℓ, r, ℓ lm, l, r	"innerste" Schleife, absolut	maximale Laufzeit, relativ	minimale Laufzeit, bezogen auf maximale	Mikroprogramm mehrfach benutzbar
Addition/Subtraktion	h	$O(n \cdot \frac{\ell}{r})$	$O(lm \cdot \frac{\ell}{r})$	9	(1)	-	nein
	v, s	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot \ell)$	14	1,6	-	ja
	v, z	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot \ell)$	11	1,2	-	ja
	a	$O(\ell)$	$O(\ell)$	-	-	-	-
Zweierkomplement	h	$O(n \cdot \frac{\ell}{r})$	$O(lm \cdot \frac{\ell}{r})$	7	(1)	-	nein
	v, s	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot \ell)$	12	1,7	-	ja
	v, z	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot \ell)$	9	1,3	-	ja
Multiplikation Boothscher Algorithmus "einfacher" Algorithmus	h	$O(n \cdot (\frac{\ell}{r})^2)$	$O(lm \cdot (\frac{\ell}{r})^2)$	20	(1)	-	nein
	v, z	$O(\frac{n}{r} \cdot \ell^2)$	$O(\frac{n}{r} \cdot \ell^2)$	15	$0,75 \cdot r$	2	nein
	v, z	$O(\frac{n}{r} \cdot \ell^2)$	$O(\frac{n}{r} \cdot \ell^2)$	10	$0,5 \cdot r$	1	nein
						2	
Division	h	$O(n \cdot (\frac{\ell}{r})^2)$	$O(lm \cdot (\frac{\ell}{r})^2)$	20	(1)	-	nein
	v, z	$O(\frac{n}{r} \cdot \ell^2)$	$O(\frac{n}{r} \cdot \ell^2)$	10	$0,5 \cdot r$	-	nein
Quadratwurzel	v, z	$O(\frac{n}{r} \cdot \ell^2)$	$O(\frac{n}{r} \cdot \ell^2)$	15	-	2	nein
						3	

Operation	horizontal/ vertikal, Standard-/ Zusatz- Hardware, assoziativ	Laufzeit abhängig von n, ℓ, r	Laufzeit abhängig von $n, \ell, r, r,$ (ml, lei)	"innerste" Schleife, absolut	maximale Laufzeit, relativ	minimale Laufzeit, bezogen auf maximale	Mikroprogramm mehrfach benutzbar
Komponentensumme	h	$O(n \cdot \frac{\ell}{r})$	$O((ml) \cdot \frac{\ell}{r})$	5	(1)	-	nein
	v, s	$O(\frac{n}{r} \cdot \ell \cdot r)$	$O(\frac{n}{r} \cdot \ell \cdot r)$	5	r	-	nein
	v, z	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot \ell)$	7	1,4	-	nein
Komponentenweiser Vergleich	h	$O(n \cdot \frac{\ell}{r})$	$O((ml) \cdot \frac{\ell}{r})$	9	(1)	$\frac{r}{\ell}$	nein
	v, z	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot \ell)$	17	1,9	$\frac{1}{\ell}$	ja
Maximum/Minimum	h	$O(n \cdot \frac{\ell}{r})$	$O((ml) \cdot \frac{\ell}{r})$	10	(1)	-	nein
	v, s	$O(\frac{n}{r} \cdot \ell \cdot r)$	$O(\frac{n}{r} \cdot \ell \cdot r)$	5	$0,5 \cdot r$	$\frac{1}{\ell}$	ja
	v, z	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot \ell)$	8	0,8	$\frac{1}{\ell}$	ja
Schwellwertvergleich	h	$O(n \cdot \frac{\ell}{r})$	$O((ml) \cdot \frac{\ell}{r})$	9	(1)	$\frac{r}{\ell}$	nein
	v, z	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot \ell)$	18	2	$\frac{1}{\ell}$	ja
Zählen addierend/subtrahierend	h	$O(n \cdot \frac{\ell}{r})$	$O((ml) \cdot \frac{\ell}{r})$	7	(1)	$\frac{r}{\ell}$	nein
	v, s	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot \ell)$	8	1,2	$\frac{1}{\ell}$	ja
	v, z	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot \ell)$	7	1	$\frac{1}{\ell}$	ja

Operation	horizontal/ vertikal, Standard-/ Zusatz- Hardware, assoziativ	Laufzeit abhängig von n, ℓ, r	Laufzeit abhängig von $n, \ell, r, m , e $	"innerste" Schleife, absolut	maximale Laufzeit, relativ	minimale Laufzeit, bezogen auf maximale	Mikroprogramm mehrfach benutzbar
"Exact Match"	h	$O(n \cdot \frac{\ell}{r})$	$O(n \cdot \frac{\ell}{r})$	7	(1)	r	nein
	v, z	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot e)$	8	1,2	$\frac{1}{r}$	nein
	a	$O(\ell)$	$O(\ell)$	-	-	$\frac{1}{r}$	-
"Best Match"	h, s	$O(n \cdot \frac{\ell}{r} \cdot r)$	$O(n \cdot \frac{\ell}{r} \cdot r)$	5	-	-	nein
	h, z	$O(n \cdot \frac{\ell}{r})$	$O(n \cdot \frac{\ell}{r})$	8	-	-	nein
	v, z	$O(\frac{n}{r} \cdot \ell \cdot \log_2 \ell)$	$O(\frac{n}{r} \cdot e \cdot \log_2 \ell)$	7	-	-	nein
Einschreiben	h	$O(n \cdot \frac{\ell}{r})$	$O(n \cdot \frac{\ell}{r})$	8	(1)	-	nein
	v, s	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot e)$	11	1,4	-	nein
	v, z	$O(\frac{n}{r} \cdot \ell)$	$O(\frac{n}{r} \cdot e)$	9	1,1	-	nein
	a	$O(\ell)$	$O(\ell)$	-	-	-	-

Nähere Erläuterung zu den einzelnen Spalten im Text

6. Die "maximale relative" Laufzeit ist das Laufzeitverhältnis der Vertikaloperationen bezogen auf die entsprechende horizontale Blockoperation. Wo beide Implementierungen gleiches Wachstum haben, ist sie das Zahlenverhältnis der "absoluten" Laufzeiten der "innersten" Schleifen; wo aber in der funktionalen Abhängigkeit ein Unterschied besteht, ist der entsprechende Faktor in den Laufzeitvergleich aufgenommen worden. Dabei sind wieder Vor- und Nachbereitung vernachlässigt.

7. Bei denjenigen Operationen oder Implementierungen, die vom Wert der Daten abhängig sind wie die bit-seriellen Multiplikationen und die Such- und Vergleichsoperationen, ist angegeben, welchen Bruchteil der maximalen Laufzeit die minimale Laufzeit für nichttriviale Fälle ausmacht (ohne Berücksichtigung von Verkürzungen durch minimale Werte von $|m|$ und $|e|$).

8. Hier ist markiert, ob es gemeinsame Mikroprogramme für verschiedene Operationen gibt, etwa Addition und Subtraktion, Maximumsuche und Minimumsuche, arithmetisch und lexikalisch.

Interpretation der Vergleiche mit Bezug auf die Strukturüberlegungen in 3.2:

1. Es ging darum, Algorithmen zu finden, die mit der notwendigen Anzahl von Zugriffen auskommen, d.h. insbesondere mit $n \cdot \ell$ Zugriffen bei linearen Operationen, $n \cdot \ell^2$ Zugriffen bei nicht-linearen Operationen vom Typ der Multiplikation, wo für jedes Operandenpaar alle Bits des einen Operanden mit allen Bits des anderen Operanden kombiniert werden müssen; eventuell muß man dabei einen konstanten Faktor in Kauf nehmen.

Als zweite Forderung an die Algorithmen hatten wir: Es sollen je r der $n \cdot \ell$ Bits simultan verarbeitet werden, möglichst mit der konventionellen Hardware. Dann reduziert sich ein Dimensionsparameter, nämlich derjenige, der sozusagen die "Parallelität" bezeichnet, um den Faktor r . Das ist für die horizontalen Blockoperationen der Parameter ℓ (Länge der Datenworte), für die Vertikaloperationen der Parameter n (Anzahl der Datenworte). Für die linearen Operationen liefert diese Reduktion für beide Implementierungen dasselbe Ergebnis; für die nicht-linearen Operationen vom Typ der Multiplikation ergibt sich ein Unterschied: Die horizontalen Blockoperationen haben um den Faktor r weniger "innerste" Schleifen als die entsprechenden Vertikaloperationen. Dabei ist

allerdings zu bedenken: Die benutzten Grundoperationen sind dabei von unterschiedlicher Komplexität; implizit enthält die Zeit etwa für eine hardwareimplementierte Multiplikation, die aus Shifts und Additionen aufgebaut ist, ihrerseits den Faktor r . Die Abschätzungen wären neu zu machen, wenn wir statt von einem Minirechner mit Parallel-Rechenwerk ausgehen von einem System aus Bit-slice-Mikroprozessoren, womöglich ohne Carry-Look-ahead-Zusatz und ohne Hardware-Multiplikation.

Ein Extremfall ist die Berechnung der Quadratwurzel: Hier haben wir einen ziemlich einfachen sequentiellen Algorithmus, der sich bit-seriell, d.h. hier: in Vertikalverarbeitung gut implementieren läßt. Eine Standard-Hardware, die ähnlich wie bei Multiplikation und Division ganze Sequenzen von Shifts und Additionen/Subtraktionen automatisiert, haben wir nicht. Deshalb wurde auf den Ansatz zu einer horizontalen Blockoperation ganz verzichtet. Die Vertikaloperation ist nicht-linear ähnlich wie die Multiplikation.

Drei scheinbar lineare Operationen fallen, wie man in der Tabelle sieht, etwas aus dem allgemeinen Rahmen: Komponentensumme, Extremumsuche (speziell: Feststellen der Abbruchbedingung), "Best Match". Bei allen drei Operationen braucht man bei einer oder beiden Datendarstellungen und Arbeitsweisen Verknüpfungen "quer" zur Sequentialisierungsrichtung. Die Komponentensumme ist ungeeignet zur Implementierung als Vertikaloperation - auf konventioneller Hardware. Es läßt sich zeigen, daß es besser geeignete Hardware gibt. Bei der Extremumsuche muß man bei der Vertikaloperation auf den Vorteil des Algorithmus verzichten, eine einfache Abbruchbedingung zu haben. "Best Match" erfordert vom Problem her eine Sequentialisierung über beide Richtungen, d.h. den Datenbestand und die Einzeldaten. Es läßt sich zeigen: Komponentensumme und Extremumsuche lassen sich durch geeignete Hardware "effektiv" implementieren, in beiden Darstellungen. "Best Match" verhält sich, mit genau demselben Zusatz-Befehl, als horizontale Blockoperation wie eine lineare Operation; als Vertikaloperation verhält sich "Best Match" wie eine nicht-lineare Operation.

2. In der Abhängigkeit von den Maskenbits unterscheiden sich die beiden Ansätze erheblich: Bei den horizontalen Blockoperationen ist der Wert des Auswahlmasken-Bit eine Bedingung, ob eine "äußere" Schleife ausgeführt werden soll;

die Bits der Relevanzmaske gehen wie Operandendaten in den Algorithmus ein. Für die Vertikaloperationen vertauschen Auswahl- und Relevanzmaske in diesem Sinne ihre Rollen. Das heißt also: Die horizontalen Blockoperationen haben überall dort günstigere Laufzeiten, wo nur auf Teilen des Datenbestandes gearbeitet wird. Die Vertikaloperationen verhalten sich dort günstiger, wo nur auf Teilen der Datenworte gearbeitet wird. Da aber gerade im Verlauf längerer Sequenzen von "assoziativen" Operationen auf immer weniger Daten gearbeitet wird, dürften trotzdem im konkreten Fall die horizontalen Blockoperationen günstiger abschneiden.

Dagegen erlaubt die Vertikalverarbeitung eine besser angepaßte Speicherung speziell der Auswahlmasken.

3. Ein wesentlicher Unterschied zwischen den beiden Ansätzen ist, daß vor allem Such- und Vergleichsoperationen bei Vertikalverarbeitung oft abgebrochen werden können, bevor alle $n \cdot l$ Bits der Gesamtinformation betrachtet worden sind. Die hier benützten Algorithmen für die horizontalen Blockoperationen erlauben solche frühzeitigen Entscheidungen im allgemeinen nicht (Ausnahme: Bei Vergleich mit einem Kriterium kann "Mißerfolg" unter Umständen schon beim höchstwertigen Teilwort festgestellt werden; man kann dann weitergehen zum nächsten Datenwort). Selbstverständlich gibt es Fragestellungen, wo auch horizontale Blockoperationen frühzeitig abgebrochen werden können: "Gibt es mindestens ein Element im Datenbestand, das mit dem Kriterium übereinstimmt?" Für die sogenannte assoziative Verarbeitung braucht man aber oft den Hinweis auf alle positiv antwortenden Elemente, weil auf ihnen allen als nächstes eine gemeinsame Operation ausgeführt werden soll.

Auch einige arithmetische Operationen verhalten sich bei Vertikalverarbeitung so datenabhängig, daß die angegebenen Abschätzungen nur für den ungünstigsten Fall zu gelten brauchen.

4. Die konventionelle Hardware begünstigt die horizontalen Blockoperationen speziell für die Vektoren-Arithmetik. Das liegt ganz einfach daran, daß in der zugrundeliegenden Maschinen-Philosophie der Einsatz für numerische Anwendungen eine so große Rolle spielt. Die Vertikaloperationen können hier nur durch Unterstützung durch Zusatz-Hardware konkurrenzfähig werden. Durch solche Zusatz-Hardware wird nichts am Grad der Parallelität des Speichers

oder des Rechenwerks gändert, sondern es wird nur das Spektrum der verfügbaren "Grundoperationen" erweitert, teils speziell auf die Vertikalverarbeitung zugeschnitten, teils allgemein verwendbar, wie oben im Zusammenhang mit "Best Match" angedeutet.

5. Über eine für die Vertikalverarbeitung günstige Speicherwort- und Rechenwerkbreite ist aus Tabelle 3.1 kein Aufschluß zu bekommen.

Es bleibt noch etwas anderes zu bedenken: Wo in den Abschätzungen " $\frac{n}{r}$ " steht, geht in Wirklichkeit ein

$$p = \left\lceil \frac{n-1}{r} \right\rceil,$$

und für " $\frac{\ell}{r}$ " steht konkret

$$q = \left\lceil \frac{\ell-1}{r} \right\rceil.$$

Im allgemeinen brauchen weder n noch ℓ ganzzahlige Vielfache von r zu sein. Wir haben dann in der Richtung der "Parallelität" Verschnitt, wobei man im allgemeinen bei Horizontalverarbeitung ℓ auf ein ganzes Vielfaches ergänzen wird, da weder der Speicherverschnitt sich ausnutzen läßt, noch die kürzere Zahl einfacher (in weniger Zeit) sich verarbeiten läßt (sondern im Gegenteil durch zusätzliche Befehle wieder verkürzt werden mußte). Wie sinnvoll es ist, bei nicht zu kleinem r mehrfach-genaue Festkomma-Arithmetik zu benutzen, soll hier nicht diskutiert werden.

6. Nicht in die Tabelle aufgenommen ist die relative Anzahl der Speicherzugriffe. Man kann davon ausgehen, daß man für die Vertikalverarbeitung bei allen Operationen, die ein Datenfeld als Ergebnis erzeugen, pro Grundverknüpfung einen Speicherzugriff mehr braucht als bei der entsprechenden horizontalen Blockoperation, um die nichtausgewählten Komponenten unbeschädigt zu erhalten - die Zusatzoperation ist das Lesen des alten Inhaltes des Bereiches, wohin das Ergebnis geschrieben werden soll. Für die nur lesenden Operationen wie alle Vergleiche und Suchoperationen haben wir die gleiche Anzahl von Speicherzugriffen für beide Verarbeitungsarten.

Wie weit sich die asynchrone Überlappung von Speicherzugriff und Verknüpfungen ausnutzen läßt, hängt vom Umfang der Verknüpfungen ab: Je weniger "gerechnet" wird, desto weniger Zeit bei den Speicherzugriffen kann man einsparen.

7. Ein Vergleich des statischen Implementierungsaufwandes ist nicht gemacht worden. Bei A. Endres /9/ findet sich die Feststellung, daß sich der Aufwand für das Beweisen von Programmen stärker als proportional vergrößert verglichen mit dem Umfang der Programme. Das bedeutet, daß unter zwei sonst gleichwertigen Mikroprogrammen nicht nur des Speicherbedarfes wegen das kompaktere zu bevorzugen ist. Teilweise geben die Skizzen einen Eindruck der Mikroprogrammlänge; dabei ist zu berücksichtigen, daß der Konstruktion "par ... end par." im konkreten Mikroprogramm keine expliziten speziellen Mikrobefehle entsprechen.

Man darf aber nicht vergessen, daß wesentliche Teile wie etwa die Parameterübernahme und -aufschlüsselung weggelassen sind und daß moderne mikroprogrammierbare Rechner auch auf Mikroprogrammebene Unterprogrammtechniken erlauben, durch die man die Mikroprogramme stark bündeln kann.

Es erscheint trotzdem betonenswert, daß die Algorithmen für die Vertikalverarbeitung - durch ihren Rückgriff zunächst nur auf die logischen Verknüpfungen - aufmerksam gemacht haben auf die Möglichkeiten, gleichwertige Aufgaben durch ein und dasselbe Mikroprogramm lösen zu lassen. Alle "Verzweigungen" werden durch Operationenmarkierungen vom Typ der "Masken" geleistet; alle solchen "Verzweigungen" sind prinzipiell dynamisch benutzbar, wie ich es allerdings nur bei der kombinierten Addition/Subtraktion für den Booth'schen Multiplikationsalgorithmus ausgenutzt habe. Die Idee solcher bedingten Operationen auf Mikrooperationsbasis geht dann auch in die Vorschläge zur Zusatz-Hardware ein.

8. Da die Standard-Hardware sehr stark auf denjenigen Befehlssatz zugeschnitten ist, den wir normalerweise "Assemblerbefehle" nennen, bestünde die Möglichkeit, die horizontalen Blockoperationen als Programmpaket aus solchen Befehlen aufzubauen. Das bedeutet kaum einen strukturellen Unterschied in den Programmskizzen; aber pro Befehl müssen wir ein bis zwei Speicherzugriffe zum Lesen dieses Befehles einkalkulieren, während die explizit aufgeschriebenen Speicherzugriffe auf die Daten zur Semantik der Befehle gehören würden. Insgesamt müssen wir mit einer Verlangsamung gegenüber den Mikroprogrammen schätzungsweise um den Faktor 2 rechnen. Bei Implementierung des Parallel- und Assoziativkonzeptes als Unterprogrammpaket werden also die Operationen - mit Ausnahme der Parallel-Multiplikation und -Division - langsamer als bei Vertikaloperation.

9. Es bleibt zu überlegen, welche qualitativen oder quantitativen Unterschiede sich ergeben beim Übergang zur *Mehrzprozessoranlage*. Sieht man einmal ab von Problemen der Datenspeicherung und des Zugriffs, die bei dezentralem Speicher auftreten, so kann man sagen: In einem System mit p Prozessoren können alle Einzeldatenoperationen vertikal mit der Parallelität $p \cdot r$ ausgeführt werden; aber auch für die Horizontalverarbeitung stehen p Rechenwerke zur Verfügung; das heißt: Die Laufzeitenverhältnisse ändern sich nicht.

Für Gesamtdatenoperationen tritt die Aufgabe auf, Information zwischen den Prozessoren auszutauschen. Die Vertikaloperation für die Maximumsuche etwa benötigt für jedes Bit-Slice die Information, ob in *allen* noch interessierenden Spalten eine Null steht oder nicht, zur Entscheidung, ob die alte Auswahlmaske für den nächsten Rechenschritt behalten werden muß oder durch einen neuen Wert ersetzt (- das hat nichts zu tun mit der Möglichkeit, einen Suchvorgang eventuell vorzeitig zu beenden). Entsprechend braucht man bei der Vertikaloperation für die Komponentensumme die Information aus *allen* ausgewählten Bits zum Aufsummieren.

Hier bleibt entweder als Lösung, alle Prozessoren auf einem je eigenen Teil des Datenbestandes unabhängig arbeiten zu lassen, etwa die Werte der Maxima in je diesem Teil des Datenbestandes zu ermitteln oder die Komponentensumme, und anschließend die Zwischenergebnisse zusammenzutragen und noch einmal zu bearbeiten - dieselbe Technik würde man bei der Horizontalverarbeitung anwenden; das heißt: Das Laufzeitenverhältnis änderte sich nicht gegenüber der Einzelprozessoranlage.

Die zweite Lösung wäre ein neuer Typ von Zusatz-Hardware, der die Kommunikation zwischen den Prozessoren erlaubte, zum Beispiel durch Weitergabe von Markierungsbits (etwa: "Ergebnis ist in allen Stellen null") von einem Prozessor zum anderen. Dann könnte man, mit einem gewissen Zusatzaufwand, für die Verarbeitung solcher Informationen, das Gesamtsystem vertikal arbeiten lassen, als hätte es ein Rechenwerk der Breite $p \cdot r$.

3.5.2 Forderungen nach Zusatz-Hardware zur Unterstützung der Vertikalverarbeitung

Die Vertikalverarbeitung erscheint strukturell besonders geeignet für die parallele Einzeldatenverknüpfung, wie in 3.2 beschrieben. Sie erlaubt eine wesentlich einfachere Maskenführung als die Horizontalverarbeitung, was vor allem in Programmen, in denen häufig Operationen bedingt auf einen Teil der Daten auszuführen sind, wichtig ist. Teilweise bietet sie einfache Algorithmen, wie zum Beispiel die Extremumsuche, die sich aber mit Standard-Hardware nicht effektiv implementieren lassen, teilweise müssen Operationen, die horizontal durch einen einzigen pseudoparallelen Befehl ausgeführt werden, in längere Sequenzen zerlegt werden. Deshalb ist zu überlegen, ob sich, ähnlich etwa der Hardware-Implementierung der Multiplikation in konventionellen Rechnern, längere Mikrobefehlssequenzen zu neuen Grundoperationen zur Unterstützung der Vertikalverarbeitung zusammenfassen lassen. In den Implementierungsskizzen sind die gewünschten neuen Mikrooperationen zum Teil bereits benutzt worden, auch in die Laufzeitabschätzungen von Tabelle 3.1 gehen sie ein. Obwohl die Laufzeitvergleiche im allgemeinen auch mit Zusatz-Hardware günstigere Zeiten für die Horizontalverarbeitung ergeben, sollen in diesem Abschnitt die gewünschten Hardware-Zusätze spezifiziert werden. Zum Schluß folgen dann Überlegungen über weitere Zeiteinsparungen durch Änderung der Speicherzugriffsstrategie.

Wieder liegt die Überlegung das Konzept der Interdata 85 zugrunde, hier vor allen Dingen ihre Busstruktur und ihr Modulkonzept.

3.5.2.1 "Vertikal-Modul"

Wir müssen davon ausgehen, daß konventionelle Rechner auch für das Mikroprogramm nur wenige logische Operationen zur Verfügung stellen - die Interdata 85 etwa nur Und, Oder, Exklusives Oder, so daß bereits die Negation "programmiert" werden muß -, daß aber gerade die typischen Vertikaloperationen aufgebaut sind aus Sequenzen von logischen Befehlen zur Implementierung von stellentreuen (d.h. komponentenweisen) Verknüpfungen. Bei der Analyse solcher Sequenzen ergibt sich dann insbesondere, daß sie meistens nicht einfach die Verknüpfung von zwei Eingangsoperanden realisieren, sondern zusätzlich im allgemeinen mindestens eine Hilfsvariable benutzen, die ein Teilergebnis des vorigen Durchlaufs der

Sequenz war, zum Beispiel das Übertragbit. Bei all denjenigen Mikroprogrammen, die mehrere Operationen oder mehrere Varianten einer Operation implementieren, wird außerdem eine "Operationenmarkierung" ("Operationenmaske") mitverknüpft.

Wir haben also: zwei Eingangsvariablen A, B, eine Ausgangsvariable S, eine "feste" Hilfsvariable D, eine Hilfsvariable C, deren neuer Wert berechnet werden muß. Um sinnvolle Markierungsbits abgeben zu können als Bedingungen für Verzweigungssprünge, zum Beispiel als Antwort auf die Abfrage, ob die Ergebnisbits in allen ausgewählten Komponenten 0 sind, brauchen wir eine Auswahlmaske M, ebenfalls als "feste" Hilfsvariable.

Als Zusatz-Hardware wird vorgeschlagen ein Modul, der alternativ zur Standard-ALU betrieben werden soll: Er soll für D, C und M Hilfsregister enthalten, die Operanden unmittelbar vom A- und B-Bus übernehmen und das Ergebnis auf den S-Bus abgeben. Abbildung 3.8. ist eine Schemaskizze eines solchen Moduls.

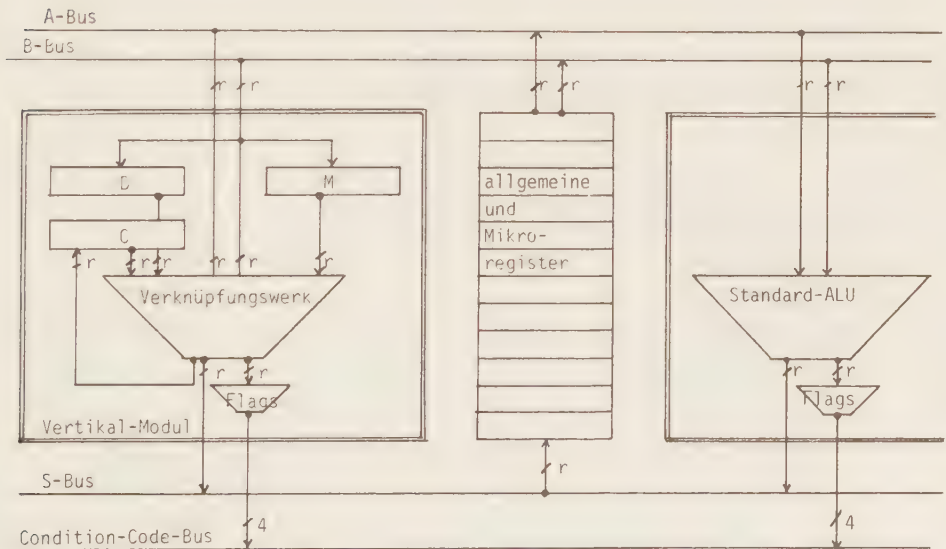


Abbildung 3.8: "Vertikal-Modul" neben Registersatz und Standard-ALU
in einem Rechner mit Modul-Konzept vom Typ der Interdata 85

Während die beiden "festen" Hilfsregister in ihrem Wert dem Mikroprogramm bekannt sind und damit etwa nach einer Unterbrechung wiederhergestellt werden können, ist der Wert von C dem Programm im allgemeinen nicht bekannt. Das Hilfsregister C muß also bei Unterbrechung im allgemeinen gerettet werden.

Bemerkung: Im Gegensatz etwa zur ~~zur~~ Hardware-Multiplikation, wenn sie aus Schiebeoperationen und Additionen aufgebaut ist, sollen hier keine iterierten Sequenzen abgekürzt werden, weil hier pro Iteration im allgemeinen mehrere Speicherzugriffe nötig sind, die nicht zu den Operationen einer Zusatz-ALU gehören können.

H. Hessenauer /19/ hat für die Hardware-Realisierung dieser Forderungen einen Entwurf gemacht, bei dem die zwei Funktionen S und C in Abhängigkeit der Werte von A, B, C, D, M durch ROMs oder PLAs implementiert werden und zwar, entsprechend dem Konzept der stellengleichen parallelen Verknüpfung, durch r identische Exemplare.

Im folgenden die Liste der Modul-Operationen, wie sie sich in 3.1.2 als notwendig erwiesen haben. a, b, c, d, m, s stehen für die Werte eines der r Bits (von A, B, C, D, M, S) und müßten, im Sinne der formalen Beschreibungen, mit j indiziert werden, wobei für jede Operation gilt

```
par   j = 0, 1, ..., r-1
      sim  sj := resultoperation (aj, bj, cj, dj, mj)
      with cj := carryoperation (aj, bj, cj, dj)
      with flags ...
      end sim
end par
```

aber c_j außerhalb des Moduls nicht unmittelbar zur Verfügung steht.

Liste der Modul-Operationen:

1. *loadregister* (*a*, *b*): lade *d* vom B-Bus;
 s irrelevant;
 c := *a*

Laden der Hilfsregister C, D

2. *loadmask* (*b*): lade *m* vom B-Bus;
 s, *c* irrelevant

Laden des Hilfsregisters M

3. *s* := *addsub* (*a*, *b*): *s* := *a* $\oplus$ *b* $\oplus$ *c*;
 c := *b* $\wedge$ *c* $\vee$ [*a* $\oplus$ *d*] $\wedge$ [*b* $\oplus$ *c*];
 Flags, ob $\bar{s} = \emptyset$ und ob $\bar{c} = \emptyset$

Addition/Subtraktion, *d* wählt Addition oder Subtraktion

4. *s* := *overflow* (*a*, *b*): *s* := *a* $\oplus$ *b* $\oplus$ *c*;
 c := *a* $\wedge$ [*b* $\oplus$ *d*] $\wedge$ $\neg$ [*a* $\oplus$ *b* $\oplus$ *c*] $\vee$ $\neg$ [*a* $\wedge$ ($\neg$ [*b* $\oplus$ *d*]) $\wedge$ (*a* $\oplus$ *b* $\oplus$ *c*)];
 Flags, ob $\bar{s} = \emptyset$ und ob $\bar{c} = \emptyset$

Berechnen der Überlaufmarkierungen (als letzter Schritt einer Additions-/Subtraktionsschleife), *d* wie bei 3.

5. *s* := *equivalence* (*a*, *b*): *s* := $\neg$ [*a* $\oplus$ *b*];
 c irrelevant;
 Flag, ob $\bar{s} = \emptyset$

Äquivalenz (zum Beispiel zur Vorzeichenberechnung beim "einfachen" Multiplikationsalgorithmus)

6. *s* := *multadd* (*a*, *b*): *s* := *a* $\oplus$ *c* $\oplus$ *b* $\wedge$ *d*;
 c := [*b* $\wedge$ *c* $\wedge$ *d*] $\vee$ [*a* $\wedge$ (*b* $\wedge$ *d* $\oplus$ *c*)]

Additionsschritt bei Abarbeitung der Multiplikatorstellen (außer Vorzeichen) beim "einfachen" Multiplikationsalgorithmus; *d* ist die aktuelle Multiplikatorstelle

7. *s* := *multsign* (*a*, *b*): *s* := *a* $\oplus$ *c* $\oplus$ *b* $\wedge$ $\neg$ *d*;
 c := [*b* $\wedge$ *c* $\wedge$ $\neg$ *d*] $\vee$ [*a* $\wedge$ (*b* $\wedge$ $\neg$ *d* $\oplus$ *c*)]

Letzter Additionsschritt bei Abarbeitung einer Multiplikatorstelle (außer der Multiplikatorvorzeichenstelle) beim "einfachen" Multiplikationsalgorithmus, leistet Korrektur für negative Multiplikanden; *d* ist die aktuelle Multiplikatorstelle

8. *s* := *multleast* (*a*, *b*): *s* := *a* $\oplus$ *b* $\oplus$ *c* $\wedge$ *d*;
 c := [*a* $\wedge$ $\neg$ [*b* $\wedge$ *d*]] $\vee$ [*a* $\wedge$ *c* $\wedge$ $\neg$ *d*]

Erster Schritt (niedrigstwertige Stelle) bei Abarbeitung des Multiplika-

torvorzeichens beim "einfachen" Multiplikationsalgorithmus; enthält Addition einer 1 zur dynamischen Erzeugung des Zweierkomplements bei negativem Multiplikator; d ist Vorzeichenbit des Multiplikators

9. $s := \text{multsub}(a, b):$ $s := a \oplus c \vee b \wedge d;$
 $c := a \wedge d \wedge c \vee \neg a \wedge (b \wedge d \oplus c)$

Weitere Schritte bei Abarbeitung des Multiplikatorvorzeichens beim "einfachen" Multiplikationsalgorithmus, entspricht Korrektur bei negativem Multiplikator; d ist Vorzeichenbit des Multiplikators

10. $s := \text{complement}(a):$ $s := a \oplus c;$
 $c := a \vee c$

Einzelschritt zur Berechnung des Komplementes (durch Vorgabe von c unterscheiden sich Einer- und Zweierkomplement)

11. $s := \text{compare}(a, b):$ $s := c \wedge \neg (a \oplus b);$
 $c := c \wedge \neg (a \oplus b);$
Flags, ob $\bar{c} = \emptyset$ und $\bar{s} = \emptyset$

Einzelschritt beim assoziativen Suchen, Vergleich unter Maske

12. $s := \text{mask}(a, b):$ $s := a \wedge \neg m \vee b \wedge m;$
 $c \text{ irrelevant}$

"Maskiertes Mischen": Einblenden von b unter Auswahlmaske m nach a

3.5.2.2 "Quersummen-Modul"

Für mehrere Vertikaloperationen (Komponentensumme, Maximum/Minimum) brauchen wir die Quersumme aller ausgewählten Bits eines Wortes. Das ist eine Operation, die sich auf konventioneller Hardware nur durch eine r-fache Schleife mit Schiebefehlern und Teiladditionen realisieren läßt. Als Markierungsbits braucht man speziell die Meldungen "keine 1", "genau eine 1", was konventionell nur durch eine Reihe von Abfragen zu gewinnen ist.

Bei der Pilot-Implementierung, die mit "Drehscheibe" arbeitet, trat eine zweite Operation auf, die "quer" zu den Speicherworten arbeitet und nur durch $O(r)$ Schiebefehle oder ein-Bit-Ausblendeoperationen implementiert werden kann: das "Packen". Unter zwei Masken sollen Bits aus zwei Operanden zusammengepackt werden; etwa sollen die ausgewählten Bit des ersten Operanden stellengerecht erhalten bleiben, die ausgewählten Bit des zweiten Operanden, unter Beibehaltung ihrer relativen Anordnung, in die "Lücken" des ersten Operanden eingeschrieben werden, soweit der Platz reicht. Eine solche Ope-

ration erschien nötig, um dann, wenn die Masken "dünn" geworden sind, den Grad an Parallelarbeit bei Vertikalverarbeitung wieder steigern zu können.

Bei einem System mit Orthogonalspeicher wäre eine solche Spezialoperation nicht nötig, weil das Zusammenpacken durch Umspeichern recht einfach zu bewerkstelligen ist. Aber auch dort treten Anwendungen auf, wo nicht ausschließlich "spaltengerecht" gearbeitet wird, sondern, zum Beispiel, der zweite Operand einer Addition "versetzt" werden muß (wie schon kurz in 3.4.1 erwähnt wurde): Bei der Schnellen Fourier-Transformation müssen in den letzten $\log_2 r$ Schritten Komponenten verknüpft werden, die jeweils innerhalb desselben Teilvektors (aus r Komponenten) stehen. Ein anderes Anwendungsbeispiel ist die Verarbeitung von Punktumgebungen von Grauwert-Mustern. Bei diesen beiden Anwendungen handelt es sich allerdings um so regelmäßige "Versetzungen", daß man das dynamisch während der Verknüpfungen notwendige Zurechtrücken auch durch Schiebeoperationen implementieren könnte.

Außer der Pack-Operation selbst brauchen wir Befehle zum Vorbesetzen der beiden Masken sowie Markierungsbits, ob die Summe der Einsen in den beiden Masken genau r ist oder größer oder kleiner.

Für die Pilot-Implementierung hat H. Steckert /53/ nach den Mikroprogrammiererwünschen einen Zusatz-Modul entworfen und gebaut für beide Funktionen: (maskierte) Quersumme und Zusammenpacken. Die Quersumme wird durch zwei ROMs für je 8 Bits und einen Addierer für 5 Bits realisiert; das Zusammenpacken wird realisiert durch getaktetes Schieben der Masken- und Datenregister, also durch eine Operation von der Art der hardware-implementierten Multiplikation in der Standard-ALU. Der Fall, daß die Maske des zweiten Operanden nirgends eine Eins hat, wo auch die des ersten eine Eins hätte, wird durch paralleles Übernehmen der Daten des ersten und zweiten Operanden beschleunigt. Wenn die zweite Maske genau die Negation der ersten ist, haben wir eine weitere Implementierung des "mask"-Befehls (vgl. Liste der Operationen des Vertikal-Moduls). Auch ein weiterer Pack-Befehl, der "linksbündig" erst alle markierten Bits des ersten Operanden übernimmt, dann die des zweiten, ist implementiert worden.

Die Abbildung 3.9 ist eine Schemaskizze, die den "Quersummen-Modul" zwischen den Bussen eines konventionellen Rechners vom Typ der Interdata 85 zeigt.

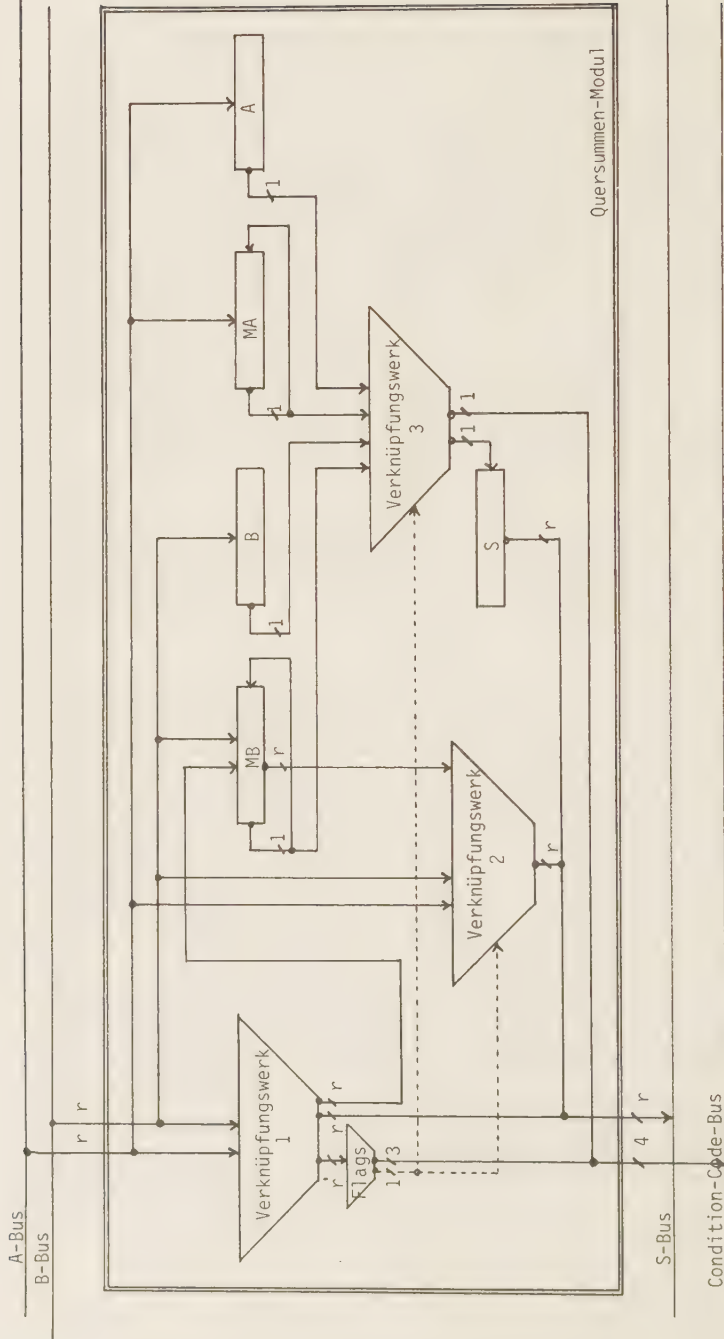


Abbildung 3.9: "Quersummen-Modul" zwischen den Bussen in einem Rechner mit Modul-Konzept vom Typ der Interdata 85

Verknüpfungswerk 1: Quersumme, Ausblenden, Vergleichen der Masken beim Laden

Verknüpfungswerk 2: Paralleles Zusammenpacken mit $m'a_j = mb_j$

Verknüpfungswerk 3: Sequentielles Zusammenpacken mit allgemeinen Masken

MA, MB sind die beiden Maskenregister, A, B sind Operandenregister (für die Pack-Befehle), S ist Ergebnisregister (für die Pack-Befehle). Der Quersummen-Befehl und der zugehörige Ausblend-Befehl arbeiten ohne Register.

Im folgenden die Liste der Modulfunktionen, wie sie sich bei der Pilot-Implementierung als notwendig erwiesen haben. a,b,ma,mb,s stehen für die Werte eines der r Bits, $\bar{s}$ ist der Wert des Ergebnisses als Binärzahl. Eine formale Beschreibung der Sequenzen soll hier vermieden werden, weil sie auch für den Mikroprogrammierer verborgen ablaufen.

Liste der Modul-Operationen:

$$1. s := \text{sum } (a, b): \quad \bar{s} := \sum_{j=\emptyset}^{n-1} a_j \wedge b_j;$$

Flags, ob $\bar{s} = 0$ oder $\bar{s} = 1$ oder $\bar{s} > 1$

Quersumme von maskierten Bits eines Speicherwortes (unter rasch veränderlicher Maske); für Komponentensumme

$$2. s := \text{collate } (a, b): \quad s_j := a_j \wedge b_j, \quad j = 0, 1, \dots, n-1;$$

$$\text{Flags, ob für } \bar{x} = \sum_{j=\emptyset}^{n-1} a_j \wedge b_j \text{ gilt}$$

$$\bar{x} = 0 \text{ oder } \bar{x} = 1 \text{ oder } \bar{x} > 1$$

($\bar{x}$ tritt außerhalb des Moduls nicht auf)

Ausblenden von maskierten Bits eines Speicherwortes, mit Information über die Anzahlen der Einsen; für Maximum/Minimum zur Entscheidung, ob weitere Vergleichsschritte nötig oder endgültige Diskriminierung möglich

$$3. \text{loadmask } (a, b): \quad s \text{ irrelevant;}$$

laden ma vom A-Bus, mb vom B-Bus;

$$\text{Flags, ob für } \bar{x} = \sum_{j=\emptyset}^{n-1} ma_j \wedge mb_j \text{ gilt}$$

$$\bar{x} = 0 \text{ oder } \bar{x} = 1 \text{ oder } \bar{x} > 1$$

($\bar{x}$ tritt außerhalb des Moduls nicht auf)

Vorbefehl zum Laden der beiden Maskenregister; entscheidet, ob sequentiell gepackt werden muß oder parallel gepackt werden darf (gilt nicht für "packleft"); bei parallelem Packen wird gesetzt: $m'a_j := \neg mb_j$ (ma_j wird "vergessen")

4. $s := \text{pack } \{a, b\}$: $s_j := a_j \wedge m a_j \vee \neg m a_j \wedge (b_k \wedge m b_k)$,
 wo b_k "auf Lücke" in freie Stellen zwischen
 die a_j eingefügt werden;
 Flag, ob nicht genügend Platz war für alle
 markierten Bits aus b

Packen der Bits des zweiten Operanden zwischen die des ersten Operanden

5. $s := \text{packleft } \{a, b\}$: $s_j := a_k \wedge m a_k \vee b_h \wedge m b_h$,
 wo die a_k "linksbündig" gepackt werden, da-
 nach die b_h "linksbündig", der Rest, falls
 vorhanden, wird mit Nullen aufgefüllt;
 Flag, ob nicht genügend Platz war für alle
 markierten Bits aus b

Linksbündiges Packen

Als Ergänzung zu dem bereits implementierten Modul:

6. $s := \text{scale } \{b\}$: $\bar{s} := \text{Anzahl der führenden Nullen in } b$;
 Flag, ob $\bar{s} = 0$

Skalierbefehl zur effektiven Abarbeitung von Relevanzmasken

3.5.2.3 Zusatz-Modul zur Verarbeitung binärer Muster

Bei den Überlegungen in 3.4.3 stellte sich bereits heraus, daß es - außer der Nutzung der schon entworfenen Spezial-Hardware - keinen Grund gibt, binäre Muster vertikal zu verarbeiten, vor allem wegen der echten Symmetrie zwischen horizontaler und vertikaler Richtung (im Gegensatz zu Codierungen von Zahlen oder Zeichen). Trotzdem sollte man den Anlaß benutzen zu überlegen, ob auch hier ein weiterer Zusatzmodul die Hardware so ergänzen kann, daß eine größere Problemklasse effektiv bearbeitet werden kann.

Vor allem für diejenigen Algorithmen, die auf Punktumgebungen arbeiten, das heißt: aus dem aktuellen Wert eines Punktes und seiner unmittelbaren vier oder acht Nachbarn den neuen Wert für den Punkt berechnen, - für diese Algorithmen erscheint es naheliegend, nach Hardware-Unterstützung zu suchen. Bei zeilenweiser Bearbeitung der zugrundeliegenden Bitmatrix brauchen wir zur Berechnung der neuen Werte in Zeile i die aktuellen Werte in den Zeilen $i-1$, i , $i+1$ und zwar sowohl stengengetreu wie auch um je eine Stelle nach rechts und links

versetzt, also etwa insgesamt, außer den Shifts, 8 logische Verknüpfungen. Es ist zu überlegen, ob sich hier ein Zusatz-Modul lohnt mit drei internen Registern für die Operandenzeilen, die zyklisch nachgeladen werden können, und einem Register für das (Zwischen-)Ergebnis, mit Shiftmöglichkeiten. Das Problem liegt in der Auswahl der Kombinationen der logischen Operationen, deren Sequenz zusammengefaßt werden soll und etwa in einem "Nanoprogramm" fixiert werden muß. Dasselbe Problem tritt auf, wenn man statt der bis zu 8 simultanen logischen Verknüpfungen der gesamten Zeilen die Sequentialisierung über die Punkte mit ihren Umgebungen wählt, also aus bis zu 9 Punkten den neuen Wert ableitet: Abgesehen davon, daß wir hier pro Zeile $O(r)$ Elementaroperationen im Modul abwickeln müssen, brauchen wir eine festgelegte Auswahl aus den möglichen Verknüpfungen von 9 binären Werten.

In dieser Allgemeinheit erscheint mir jedenfalls ein solcher Modul nicht implementierbar.

3.5.2.4 Spezieller Speicherprozessor

Bei jedem mikroprogrammierten Rechnersystem kommen wir auf das alte Konzept der zwei getrennten Speicher zurück: ein Speicher für das Programm - hier das Mikroprogramm - und ein zweiter für die Daten, zu denen auch die emulierten Makrobefehle gehören. Bei datenintensiven Anwendungen ist es dann oft die Geschwindigkeit des Datenspeichers, die diejenige des Gesamtsystems bestimmt. Bei unserem mikroprogrammierten Parallel- und Assoziativkonzept haben wir zwar vermutlich weniger Zugriffe auf Benutzerbefehle (sieht man einmal von den Zugriffen auf die Versorgungsblöcke usw. ab), aber sehr viel mehr Zugriffe auf die Daten im engeren Sinne als bei akkumulator- oder register-orientierten konventionellen Systemen, wo man davon ausgehen kann, daß ein erheblicher Anteil der berechneten Werte unmittelbar weiterverwendet wird zur nächsten Rechnung, und nicht im Hauptspeicher abgespeichert wird. Für eine zweistellige Vertikaloperation wie etwa die Addition braucht man pro Bit-Slice-Verknüpfung vier Speicherzugriffe: Lesen der beiden Operanden, Lesen des alten Inhaltes im Ergebnisbereich, Schreiben des einmaskierten neuen Ergebnisses. Beim Orthogonalspeicher, wie ihn H. Hessenauer /19/ vorschlägt, könnte man unter Umständen auf das Lesen aus dem Ergebnisbereich verzichten und maskiert einschreiben.

Für die Abschätzungen in 3.5.1 wurden folgende Festlegungen zugrunde gelegt: Logische Operationen, Schiebeoperationen, Addition, Subtraktion, Sprünge brauchen eine Zeiteinheit; Speicherzugriffe brauchen zwei Zeiteinheiten: Diese "2" ist eine statistische Verallgemeinerung; die Zykluszeit bei der Interdata 85 für den Hauptspeicher ist ungefähr 2 mal so groß wie die Zykluszeit für den Mikroprogrammspeicher; in günstigen Mikrobefehlssequenzen kann aber eine Speicheroperation von einem beliebigen Mikrobefehl angestoßen werden, verbraucht also keinen explizit auftretenden Befehl, während in anderen Fällen Befehle zum Erzeugen und Laden von Adressen nötig sind.

In Tabelle 3.2 ist für die drei Implementierungsskizzen für die Vektorenaddition (aus 3.4.1 und Anhang A.1) und ein Konzept mit Orthogonalspeicher, in den man maskiert schreiben kann, angegeben, wieviele verknüpfende Mikrobefehle die innerste Schleife enthält, einschließlich von 2 Befehlen für die Schleifenorganisation, wieviele Speicherzugriffe eine Schleife enthält und - für zwei verschiedenen Zugriffszeiten, nämlich "2" und "1" Takteinheiten - die Gesamtzeit für die Schleife und der relative Anteil davon für die nicht mit anderen Operationen überlappenden Speicherzugriffszeiten.

Wie man sieht, fällt bei der "rechenintensiven" Vertikaloperation auf der Standard-Hardware, die nackte Speicherzeit am wenigsten ins Gewicht. Oder anders ausgedrückt: Bei den Implementierungen mit Zusatz-Hardware und der horizontalen Blockoperation äußert sich eine Verkürzung der Speicherzugriffszeit viel spürbarer.

Für Operationen mit nur einem Operanden, also im wesentlichen mit nur einem Speicherzugriff in der innersten Schleife, gelten ähnliche Abschätzungen.

Im folgenden sollen zwei Lösungskonzepte skizziert werden: Zunächst soll ein Spezialprozessor am Speicher vorausgesetzt werden, von dem vor allem die "regelmäßigen" Speicherzugriffe automatisch ausgeführt würden, einschließlich der Adreßerzeugung. Auf jeden Fall müssen aber auch "normale" ("unregelmäßige") Speicherzugriffe - zum Beispiel auf Befehlsfolgen mit Sprüngen - einfach abgewickelt werden können.

Implementierungsansätze für Vektorenaddition	Anzahl der Verknüpfungen und Schleifen- organisation	Anzahl der Speicher- zugriffe	Speicherzugriff: 2 Zeiteinheiten		Speicherzugriff: 1 Zeiteinheit	
			Gesamtzeit	nicht überlappte Speicherzeit	Gesamtzeit	nicht überlappte Speicherzeit
horizontal	1 + 2	3	9	$\frac{6}{9} = 0,67$	6	$\frac{3}{6} = 0,5$
vertikal, Standard-Hardware	9 + 2	4	14	$\frac{3}{14} = 0,19$	12	$\frac{1}{12} = 0,08$
vertikal, Zusatz-Hardware	2 + 2	4	11	$\frac{7}{11} = 0,64$	7	$\frac{3}{7} = 0,43$
vertikal, Zusatz-Hardware, Orthogonalspeicher	1 + 2	3	9	$\frac{6}{9} = 0,67$	6	$\frac{3}{6} = 0,5$

1
93
1

Tabelle 3.2: Vergleich der gesamten Laufzeiten und der expliziten Speicherzugriffszeiten bei der Vektorenaddition - verschiedene Implementierungen und unterschiedliche Speicherzugriffszeiten

Die nicht mit anderen Operationen überlappte Speicherzugriffszeit ist angegeben als Bruchteil der Gesamtlaufzeit (aus Abschätzungen der "innersten" Schleife der zugehörigen Mikroprogramme)

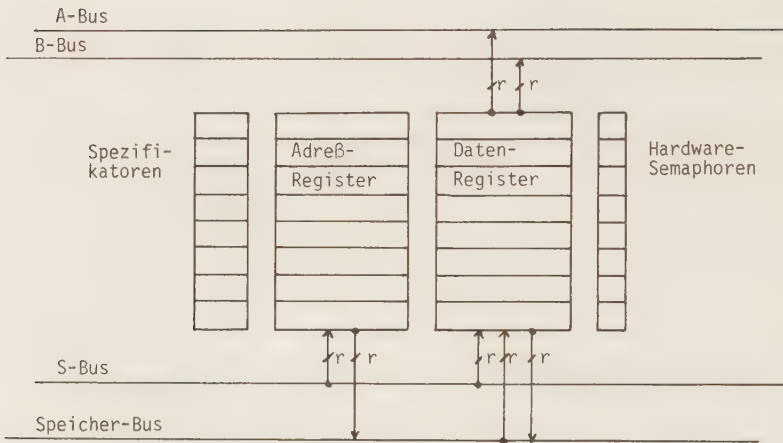


Abbildung 3.A: Adreß- und Datenregister zwischen Zentral- und Speicherprozessor

Spezifikatoren: lesen/schreiben,
Adressen aufwärts/abwärts fortschalten

Hardware-Semaphoren: markieren, welcher Prozessor als nächster
auf das Datenregister zugreifen muß; ver-
riegeln gegen den Zugriff des anderen

Zwischen Zentral- und Speicherprozessor liegt ein Satz von Registerpaaren (vgl. Abbildung 3.A) und zwar einander zugeordnete Adreß- und Datenregister. Zu jedem Registerpaar gehören Spezifikatoren - ob gelesen oder geschrieben werden soll, ob im Adreßregister automatisch aufwärts oder abwärts weitergezählt werden soll. Außerdem gibt es pro Registerpaar einen Hardware-Semaphor, der dafür sorgt, daß die beiden Prozessoren genau abwechselnd auf das zugehörige Datenregister zugreifen, und der dem Speicherprozessor zur Markierung dient, welche Registerpaare zu bedienen sind (das spart eine zusätzliche "aktiv/inaktiv"-Markierung).

Durch einen speziellen Vorschaltbefehl (etwa über einen Zusatz-Modul zur Speicherzugriffsvorbereitung) soll ein Adreßregister mit seinen Spezifikatoren geladen werden, unabhängig vom Stand des Semaphors: Der Semaphor wird dabei so initialisiert, wie der Lese-/Schreib-Spezifikator angibt. Bei "Lesen" (aus dem Speicher) muß der Speicherprozessor den ersten Zugriff tun, bei "Schreiben" (in den Speicher) der Zentralprozessor. Die weiteren Zugriffe

sollen dann automatisch abgewickelt werden, d.h. abwechselnd von beiden Prozessoren mit automatischem Fortschalten der Speicheradresse. Der Aufwand für das Aufsetzen ist gering.

Die Datenregister sollen, wie allgemeine und Mikroregister, einfach adressierbar sein als als Quellen- und Zielregister aller Mikrobefehle. Das bedeutet natürlich konkret, daß der Adressierraum für alle Register erweitert werden muß bzw. einige Mikroregister geopfert. Aber auch im Standard-Konzept der Interdata 85 treten das Speicheradreß-, das Speicherdaten-, das Befehlszähler- und das Befehlsregister mehr oder weniger explizit mit speziellen Registeradressen auf.

Man könnte sich etwa einen Satz von vier Registerpaaren vorstellen, wovon eines Befehlszähler und Befehlsregister sind. Bei einem Drei-Adreß-Befehl vom Typ der Vektoren-Addition enthalten zwei Registerpaare die Operandenadressen und -werte, eines Ergebnisadressen und -werte, wobei vorausgesetzt ist, daß beim Schreiben in den Speicher maskiert wird.

Da das Mikroprogramm die "Basisadressen" kennt und, etwa durch einen Schleifenzähler, indirekt auch die aktuellen Adressen, kann es nach einer Unterbrechung wieder korrekt aufsetzen.

Der "Normalbetrieb", d.h. ein Programmstrom, bei dem die Daten nicht konsekutiv aus dem Speicher geholt bzw. dort abgelegt werden, könnte mit nur zweien der Registerpaare arbeiten und hätte keinen größeren Overhead als in der Standard-Ausrüstung. Allerdings dürfte das Daten-Register, im Gegensatz zur Standard-Ausrüstung, nicht zweckentfremdet und gelegentlich als allgemeines Hilfsregister benutzt werden.

In Tabelle 3.3 sind, entsprechend Tabelle 3.2, für zwei verschiedene Speichergeschwindigkeiten Abschätzungen angegeben für die gesamte "Rechenzeit" sowie das Verhältnis "Speicherzeit" zu "Rechenzeit". Man sieht, wo bei den verschiedenen Ansätzen die Limitierung liegt. Für die "ausgeglichene" Lösung haben wir hier in dem speziellen Beispiel: Zwei Speicher mit gleicher Geschwindigkeit, einer für die Daten, der andere für das (Mikro-)Programm.

Die gesamten Auszählungen und Abschätzungen gehen davon aus, daß der (Datei-) Speicher jederzeit verfügbar ist. Im realistischen Fall muß man aber damit rechnen, daß nebenher zum Beispiel Datenkanäle arbeiten oder daß, in einem

Implementierungsansätze für Vektorenaddition	Anzahl der Rechen- operationen	Anzahl der Speicher- zugriffe	Speicherzugriff: 2 Zeiteinheiten			Speicherzugriff: 1 Zeiteinheit		
			Gesamt- zeit	Verhältnis Speichern/ Rechnen	Limitierung	Gesamt- zeit	Verhältnis Speichern/ Rechnen	Limitierung
horizontal	3	3	6	2	Speichern	3	1	-
vertikal, Standard-Hardware	11	4	11	0,72	Rechnen	11	0,36	Rechnen
vertikal, Zusatz-Hardware	4	4	8	2	Speichern	4	1	-
vertikal, Zusatz-Hardware, Orthogonalspeicher	3	3	6	2	Speichern	3	1	-

Tabelle 3.3: Vergleich der gesamten Laufzeiten und der Zeitverhältnisse von Berechnungszeit zu Speicherzugriffszeit bei der Vektorenaddition - verschiedene Implementierungen in einem System mit zusätzlichem Speicherprozessor

Die Abschätzungen sind wieder für die "innersten" Schleifen der zugehörigen Mikroprogramme angegeben. Die "Zeiteinheiten" sind etwas größer als in einem einfachen System (d.h. ohne speziellen Speicherprozessor).

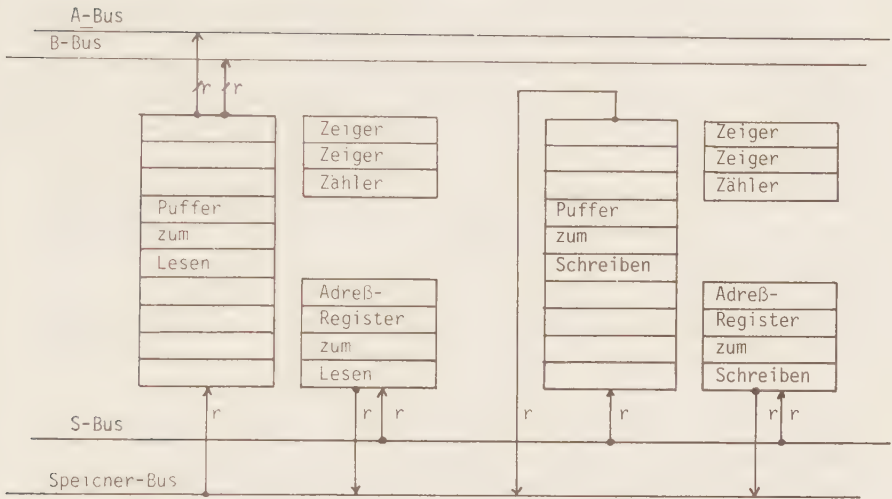


Abbildung 3.B: Puffer zum Lesen und Schreiben zwischen Zentral- und Speicherprozessor

Zeiger: zeigen auf Anfang und Ende des Pufferbereiches; Anfang: für den jeweils in den Puffer schreibenden Prozessor; Ende: für den jeweils aus dem Puffer lesenden Prozessor; Gleichheit der Zeiger markiert Ausnahmesituation (Puffer leer oder voll)

Zähler: zählt, wie oft Einzelauftrag abgewickelt wurde, etwa Lesen alternierend aus zwei Operandenbereichen

Mehr-Prozessor-System, gerade ein anderer Prozessor auf den Speicher zugreift. Das bedeutet dann: Reduktion der Arbeitsgeschwindigkeit für das Mikroprogramm. Durch eine "Pufferung" der Speicherzugriffe beziehungsweise der Daten kann man diese Geschwindigkeitsreduktion teilweise verhindern: Außerhalb der sehr speicherintensiven "innersten" Schleifen der meisten Mikroprogramme stehen mehr oder weniger lange Mikrobefehlssequenzen ohne Datenspeicherzugriffe. Setzt man zwischen Spezialprozessor und Speicherprozessor zwei genügend lange Warteschlangen (FIFO-Speicher, "first in, first out") zum Lesen und zum Schreiben von Daten und sorgt außerdem für eine "Automatik" der Adreßerzeugung, kann man mit dem Lesen etwa schon weit vor Beginn der "innersten" Schleife anfangen und das Schreiben auf den Rest des Programmes verschieben. Bei einem solchen Konzept gehört zum Initialisieren außer den Spezifikatoren auch die Angabe, welche Datenbereiche in welcher Reihenfolge zu bedienen sind. Um die Warteschlangen

nicht mit nicht mehr gebrauchten Daten zu füllen, sollten sie außerdem einen Zähler haben, wie oft sie die Folge der bestellten Daten geliefert haben, beziehungsweise, wieviele der abgelegten Daten im angegebenen Bereich einzuschreiben sind, siehe Abbildung 3.B.

Schwierig ist hier das Einbetten des "Normalbetriebes" und das Wiederaufsetzen nach Unterbrechung, unter Umständen auch das Anstoßen einer später folgenden Sequenz, bevor die Schlange leer ist.

Beide Konzepte - adressierbare Adreß- und Datenregister und Warteschlangen - sind nicht durch einfache "Erweiterung", "Ergänzung" des Standard-Konzeptes zu erreichen, und zwar unabhängig davon, ob die Schnittstelle zwischen Zentral- und Speicherprozessor für den Zentralprozessor ein "Modul" oder eine Peripherie-Schnittstelle ist. Vor allem der Wunsch, die Datenregister oder die Köpfe der Warteschlangen als Quellen und Ziele aller verknüpfenden Mikrobefehle benützen zu können, läßt sich nicht erfüllen ohne tiefer greifende Einbettung der Register oder Warteschlangenzeiger in das Zentralsystem.

4. Entwicklung von Graphenalgorithmen für Parallel- und Assoziativrechner. Vergleich für verschiedene Rechnerkonzepte

Beim Entwurf der Befehlsliste für das mikroprogrammierte Parallel- und Assoziativkonzept wurde versucht, mögliche Anwendungen zu berücksichtigen bei Auswahl und Konstruktion der Operationen. Dieses Kapitel sollte ursprünglich sozusagen die Rückkopplung liefern: Typische Anwendungsbeispiele aus verschiedenen Bereichen sollten Konsistenz und Leistungsfähigkeit des Konzeptes zeigen, aber auch weitere Forderungen an das Konzept stellen. Unter Umständen sollten sich Kriterien ergeben, ob eine der beiden Implementierungen zu bevorzugen wäre. Außerdem interessiert auch auf diesem Niveau der Vergleich mit hardware-implementierten Konzepten.

Bereits die Zeitvergleiche in Kapitel 3 ließen es wenig ergiebig erscheinen, nach numerischen Anwendungsbeispielen zu suchen, die für die Vertikalverarbeitung sprächen. In einem ersten Abschnitt habe ich drei numerische Beispiele zusammengestellt, die mir typisch erscheinen.

Dann aber habe ich mich konzentriert auf ein Anwendungsgebiet, bei dem man sowohl auf die Parallel- wie auf die Assoziativarbeit zurückgreifen kann und wo zudem teilweise vergleichbare Spezialhardware existiert: Algorithmen für Graphen. Dabei ergab sich eine allgemeinere Studie über die Lösung von Graphenproblemen mit unkonventionellen Rechnerkonzepten. Zugleich klärte sich der Unterschied zwischen Parallel- und Assoziativkonzept (die zum Beispiel beim STARAN dasselbe zu sein scheinen): Vielfachoperationen adressierbarer Datenblöcke und Vielfachoperationen mit inhaltlicher Identifizierung der Daten.

Weitere Anwendungsgebiete wie etwa alphanumerische Listenverarbeitung und Bildverarbeitung sind hier nicht behandelt worden.

4.1 Kurze Zusammenfassung von einigen numerischen Anwendungsbeispielen

1. Ein typisches Beispiel, bei dem sehr intensiv mit Vektorarithmetik gearbeitet werden kann, ist *Fast-Fourier-Transform*. Für $N=2^n$ Stützstellen brauchen wir n Iterationsschritte, die prinzipiell alle vom gleichen Typ sind: Im Iterationsschritt s ($s = 1, 2, \dots, n$) werden s Vektorenpaare mit je 2^{n-s} Komponenten verarbeitet und zwar mit Konstantenvektoren multipliziert und anschließend

addiert bzw. subtrahiert. Ein Konzept mit Vektorenarithmetik für parametrisierbare Komponentenzahl erscheint hier bestens geeignet. Bei näherem Zusehen zeigt sich, daß die Vertikalverarbeitung ein solches Konzept *nicht* ist: Wenn $n < r$ ist, ist es nicht mehr möglich, Komponentenpaare, die verknüpft werden sollen, aus derselben Spaltenspalte zu entnehmen; hier müßte also erstens das Prinzip der Vertikalverarbeitung erweitert werden so, daß auch nicht spaltengetreue Verknüpfung möglich ist, und zweitens müßte der Algorithmus bei $s \leq n-r$ "umgeschaltet" werden von Operationen mit spaltengetreuen Verknüpfungen von Operanden, die in zwei verschiedenen Speicherbereichen (Bitmatrizen) stehen, auf Operationen mit nicht-spaltengetreuen Verknüpfungen von Operanden aus ein und demselben Speicherbereich. Dabei treten zusätzliche Probleme auf, wenn r nicht eine Zweierpotenz ist, was ja heutzutage doch noch für einige Rechner gilt.

Daß für Vektorenarithmetik die Konstanten ebenfalls als Vektoren abgespeichert sein müssen, das heißt insbesondere: daß viele der Konstantenwerte vielmals abgespeichert sein müssen, haben wir bei jedem anderen Vektorkonzept ebenfalls, ebenso die daraus entstehenden Schwierigkeiten, die Algorithmen in n (das heißt: der Zahl der Stützstellen) und ℓ (der Datenlänge) zu parametrisieren.

Es erscheint aber als spezielle Einschränkung der Vertikalverarbeitung, daß wir nur über Festkomma-Arithmetik verfügen. Die frei wählbare Datenlänge ℓ wiegt diesen Nachteil wohl kaum auf.

Es ist also insgesamt der Schluß zu ziehen: Um Fast Fourier Transform auf einem konventionellen Universalrechner effektiv auszuführen, ist die konventionelle Emulation vorzuziehen (mit einem Operationensatz, der die Gleitkomma-Arithmetik enthält), das heißt: der konventionelle sequentielle Algorithmus, bei dem ein großer Teil der Schleifenorganisation im Mikroprogramm ausgeführt wird.

2. In Anhang B ist ein verblüffender Algorithmus nach /29/ beschrieben, mit Hilfe von gleichartigen parallel arbeitenden Prozessoren die n -te Potenz einer vorgegebenen Zahl zu berechnen; dieser Algorithmus ist besonders elegant, wenn die Anzahl r der Prozessoren nicht kleiner als n ist. Der Algorithmus besteht aus einer Vektorensubtraktion, einer Vektorendivision, einer Aufsummierung der Komponenten - in /29/ aus $\lceil \log_2 n \rceil$ nichtspaltengetreuen Vek-

torenadditionen, bei uns aus einer Komponentenaufsummierung als Spezialoperation -, einer "einfachen" Division und einer "einfachen" Addition (Verknüpfungen von Einzelzahlen). Neben der Spezialoperation für die Komponenten-summe erweist sich hier als nützlich, daß wir das konventionelle Rechnerkonzept mit seiner Befehlsliste zur Einzeldatenverknüpfung kombinieren können mit dem mikroprogrammierten Parallelkonzept.

Auch hier muß die Entscheidung zwischen den beiden mikroprogrammierten Implementierungen gegen die Vertikalverarbeitung fallen, weil wir für nichttriviale Daten (Ausgangszahl und Exponent) so erhebliche Größenunterscheide der Zwischenergebnisse erhalten, daß es sich im allgemeinen empfiehlt, mit Gleitkommazahlen zu rechnen.

3. Es gibt aber auch numerische Probleme, wo sich die Kombination aus Parallel- und Assoziativkonzept als besonders nützlich erweist. Als typisches Beispiel sei die *Multiplikation sehr dünner Matrizen* betrachtet. Die bekannten Speichertechniken /44/ für dünne Matrizen in konventionellen Rechnern arbeiten entweder mit einem starren Anordnungsschema, bei dem es sehr schwer wird, neue Elemente einzufügen, oder aber es wird schwer, zusammengehörige Elemente (z.B. Zeilen oder Spalten der Matrix) schnell zuzuordnen. Der assoziative Datenzugriff erlaubt uns, bei geeigneter Abspeicherung des Index zu einem Element diese Schwierigkeiten zu vermeiden.

Für die Multiplikation der Zeilen mit je einem Element nutzen wir aus, daß wir Vektoren beliebiger Komponentenzahl verarbeiten können; beim Aufaddieren müssen wir zwar sequentialisieren, haben aber dafür den einfacheren Zugriffsmechanismus.

Hier erscheinen die beiden mikroprogrammierten Implementierungen unseres Parallel- und Assoziativkonzeptes als gleichwertig.

4.2 Algorithmen für Graphen

In den folgenden Abschnitten sind eine Reihe von Algorithmen für Graphen skizziert, wie sie auf einem Rechner mit dem (mikroprogrammierten) Parallel- und Assoziativkonzept (Kapitel 3) implementiert werden können. Die Motivation, sich mit

solchen Algorithmen zu beschäftigen, kam zunächst durch die möglichen Anwendungen wie etwa die Gauss-Elimination dünner Matrizen /3/ und Probleme der Ablaufsteuerung /35/. Hinzu kam der Ehrgeiz, zu versuchen, mit Spezial-Hardware wie assoziativem Speicher /6/ und zellularen Arrays /31/ zu konkurrieren. Diese Algorithmen erweisen sich dabei als typische Anwendungsbeispiele für unser "gemischtes" Parallel- und Assoziativ-Konzept. Man kann aber auch insbesondere die Grenzen solcher Konzepte dabei erkennen.

Graphenalgorithmen werden oft klassifiziert als *enumerativ* und als *algebraisch*, d.h. als Algorithmen, bei denen die gesuchten Lösungen nacheinander gefunden werden, etwa durch Probieren, und solche, bei denen eine geschlossene Formel angegeben werden kann, nach deren Rechenvorschrift die gesuchten Lösungen erzeugt werden. Im allgemeinen setzen die beiden Algorithmenklassen verschiedene Darstellungen des zugrundeliegenden Graphen voraus: Die enumerativen Algorithmen arbeiten auf Dateien, die algebraischen zum Beispiel auf Matrizen. Speziell Algorithmen der ersten Klasse eignen sich zur Implementierung auf allgemeinen assoziativen Systemen, Algorithmen der zweiten Klasse können eventuell auf zellularen Arrays implementiert werden. Allzu scharf lassen sich die beiden Klassen aber nicht voneinander trennen.

Stillschweigende Voraussetzung für alle Algorithmen ist: Es gibt keine mehrfachen Kanten.

Ich werde vier Gruppen von Algorithmen untersuchen. Die Auswahl ist in einem gewissen Sinn zufällig, die Gliederung ist thematisch, berücksichtigt aber auch, ob etwa eine Implementierung durch ein zelluläres Array möglich ist oder aber der Algorithmus zu komplex dafür ist. Wo mir zelluläre Arrays bekannt sind, habe ich sie kurz erwähnt, soweit sie nicht sogar in 4.2.1 als explizite Beispiele zitiert sind. Bei den Eliminationsgraphen bin ich von Algorithmen für konventionelle Rechner ausgegangen und habe versucht, sie auf das Parallel- und Assoziativkonzept anzupassen und, als Alternative, sie für dieses Konzept neu zu formulieren. Zum Schluß habe ich versucht, für ein komplexeres Problem eine adäquate Lösung auf dem Parallel- und Assoziativkonzept zu finden: Beim Algorithmus zur Bestimmung des kürzesten Hamilton-Weges zeigen sich recht deutlich die allgemeinen Schwierigkeiten jedes Parallel- und Assoziativkonzeptes.

Für alle Algorithmen sollen *abstrakte* Laufzeitabschätzungen angegeben werden, das soll heißen: Abschätzungen durch Auszählen der Laufschleifen ohne Rücksicht auf Daten- und Datenlängeabhängigkeiten. Zum Schluß des Kapitels sollen diese Abschätzungen verfeinert werden, um das *konkrete* Laufzeitverhalten für verschiedene Algorithmen und verschiedene Rechnerkonzepte vergleichen zu können.

4.2.1 Vergleichbare unkonventionelle Spezial-Hardware: Zellulare Arrays

Viele Algorithmen in den folgenden Abschnitten wurden angeregt durch eine Arbeit über zellulare Arrays zur Lösung von Graphenproblemen /31/. Es soll sich zeigen, daß die veröffentlichten Algorithmen im wesentlichen nachgebildet werden können. Dabei sind gelegentlich explizite Sequentialisierungen nötig, wo dort höchstens Laufzeiten von Signalen erwähnt und berücksichtigt werden müssen. In unserem Konzept wird der Verlust an Geschwindigkeit ausgeglichen durch Flexibilität: Die Knotenzahl n braucht hier nicht fest zu sein; wir sind nicht eingeschränkt auf eine oder wenige hardware-implementierte Funktionen; die Daten können beliebig lang sein und beliebig strukturiert werden. Damit existieren hier erheblich weitergehende Möglichkeiten als bei den zellularen Arrays. Dies soll sich später in detaillierten Beispielen zeigen.

Die zugrundeliegenden Algorithmen gehören alle zur Kategorie der "algebraischen" Verfahren.

Kurze Beschreibung des allgemeinen Schemas der hier zitierten zellularen Arrays: Es handelt sich um einen Spezial-Feldrechner. Er besteht als zweidimensionales schachbrettartiges Feld aus identischen Moduln (Zellen), wobei jede Zelle wenige Bits Flip-flop-Speicher und ein bißchen kombinatorische Logik enthält und nur mit ihren vier unmittelbaren Nachbarn im Feld Information austauscht. Die einzigen Ein- und Ausgaben für das zellulare Array sind Verbindungen zu den Zellen an den Array-Rändern, einschließlich einiger Signalleitungen, die durch die Reihen oder Spalten geführt sind (zitiert nach /31/). Die Zellen des Arrays speichern etwa die Information der Adjazenzmatrix A . Für die verwendbaren Algorithmen ist zu bedenken: Es sollen möglichst Operationen, die in einem sequentiellen Algorithmus nacheinander ausgeführt werden, zusammenfaßbar sein (zum Beispiel die Feststellung mehrerer Auswahlbedingungen). Die Zellen arbeiten asynchron. Die Möglichkeiten zu assoziativen Suchvorgängen

sollen ausgenutzt werden. Das Verhältnis zwischen der Komplexität der einzelnen Zelle und der Rechengeschwindigkeit muß berücksichtigt werden. Zur Veranschaulichung sollen zwei unterschiedlich komplizierte zellulare Arrays aus [31] hier skizziert werden (Abbildungen 4.1 und 4.2).

1. Das erste zellulare Array testet, ob ein Graph (ohne mehrfache Kanten) stark zusammenhängend ist. Die Zellen enthalten zu Beginn die Werte der Elemente der Adjazenzmatrix. Ausgegeben wird ein Testsignal t , ob der zugehörige Graph stark zusammenhängend ist oder nicht. Abbildung 4.1 ist die Schemaskizze dieses Arrays.

Die Laufzeit ist $O(n^2)$, wenn nur Verzögerungen für die Gatter (nicht für die Leitungen) angenommen werden.

2. Abbildung 4.2 ist die Schemaskizze einer Zelle $C_{i,j}$ des zellularen Arrays zur Bestimmung eines aufspannenden Baumes. Hier brauchen wir am Rand ein Hilfsarray, speziell zur Auswahl der ersten Zelle (genauer des ersten Index i), die alle Auswahlbedingungen erfüllt.

Sollen außerdem die Fundamentalzyklen zum Baum T bestimmt werden, so muß die Zelle $C_{i,j}$ erweitert werden; wir brauchen außerdem Schieberegister zur Auswahl von weiteren Knoten für Fundamentalzyklen und zum Festhalten des zuletzt gewählten Knoten im Baum. Zusätzlich brauchen wir eine Ausgabemöglichkeit, um jeden einzelnen der Fundamentalzyklen, nachdem er bestimmt wurde, auszugeben. Hier wird nicht nur die Kante aus dem Co-Baum angegeben, sondern der ganze Zykel. Es müssen also außerhalb des Arrays weitere Rechnerkomponenten arbeiten zur Steuerung des Ablaufs und zur Übernahme der Zwischenergebnisse.

Für die Laufzeit ist angegeben: " $n-1$ Iterationen". Es fehlt aber die Angabe der Komplexität für die einzelne Iteration.

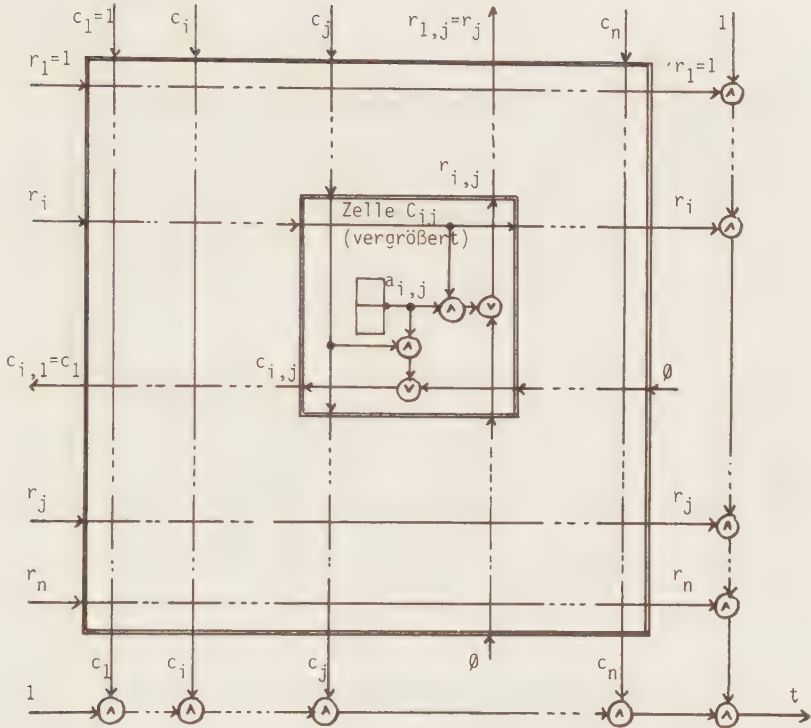


Abbildung 4.1: Zellulares Array zum Test auf starken Zusammenhang
(nach /31/)

Es gelten folgende funktionalen Zusammenhänge:

$$c_{i,j} = c_j \wedge a_{i,j} \vee c_{i,j+1}; \quad r_{i,j} = r_i \wedge a_{i,j} \vee r_{i+1,j};$$

$$c_i = \bigvee_{j=1}^n c_j \wedge a_{i,j}; \quad r_j = \bigvee_{i=1}^n r_i \wedge a_{i,j}.$$

Das Testsignal ist genau dann 1, wenn der Graph mit der Adjazenzmatrix $A = (a_{i,j})$ stark zusammenhängend ist.

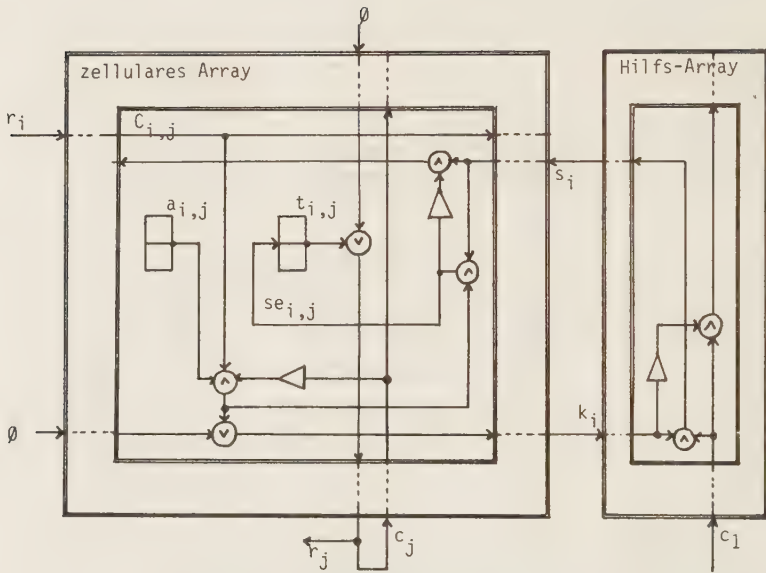


Abbildung 4.2: Zellulares Array zur Bestimmung eines aufspannenden Baumes (nach /31/)

Es gelten die Beziehungen: $c_{i,j} = a_{i,j} \wedge r_i \vee \neg c_j$;

$$k_i = \bigvee_{j=1}^n c_{i,j}.$$

s_i wählt den Knoten v_i an, der unter allen Konkurrenten die höchste Nummer i hat. Eine entsprechende Auswahl unter den Knoten v_i findet in den Zellen $C_{i,j}$ ($j = 1, 2, \dots, n$) dann statt zur Bestimmung von $se_{i,j}$ und damit $t_{i,j}$.

4.2.2 Notation

B. Jahn hat in seiner Diplomarbeit /25/ eine Algol 68-Erweiterung konzipiert, die es erlaubt, die in Kapitel 3 entworfenen Befehlsklassen, etwa die assoziativen Befehle, dem Programmierer einer höheren Programmiersprache nutzbar zu machen. Die neuen Sprachkonstruktionen sind von der bei Algol 68 gewohnten Komplexität. Für die Skizzierung der Algorithmen-Beispiele hier genügt aber eine weniger programmspezifisch formalisierte Notation, da der wesentliche Zweck dieser Skizzen sein soll - außer der korrekten Beschreibung des Algorithmus -, die Kontrollstrukturen klar erkennen zu lassen insbesondere für Laufzeitabschätzungen.

Eine andere Möglichkeit - neben dem Entwerfen einer eigenen Notation - wäre gewesen, auf eine existierende Graphen-Sprache zurückzugreifen, wie sie U. Pape in einem Übersichtsartikel /39/ zusammengestellt hat. So erscheint mir etwa GRAAL (vgl V. Basili /1/) gut geeignet, Datei-Algorithmen zu programmieren, während diese Sprache keine Möglichkeiten enthält, Feld-Algorithmen konsistent zu programmieren. Daß entsprechende Erweiterungen von GRAAL vorgesehen sind bzw. im Rahmen einer größeren Sprachfamilie bereits möglich, ist der Veröffentlichung von V. Basili /1/ zu entnehmen. Außerdem bräuchte man aber zusätzliche Sprachelemente, um explizit zwischen sequentieller und paralleler Ausführung von Anweisungen unterscheiden zu können. Die anderen von U. Pape zitierten Graphen-Sprachen sind noch weniger geeignet, weil sie entweder in ihrer Datenstruktur oder im Vorrat ihrer Steuerbefehle zu stark eingeschränkt sind oder in beidem. Die Sprache GRAMAL /41/ ist aus denselben Gründen nicht geeignet.

1. Auch auf dem Benutzerniveau brauchen wir Konstruktionen, die es erlauben, parameterisiert Mehrfachoperationen zu beschreiben und zwar entweder sequentiell wiederholte Operationen oder parallel ausgeführte Operationen. Wie in Kapitel 3 benutzen wir dazu die Konstruktionen sequ ... end sequ und par ... end par.

Parameter der Mehrfachoperationen sind entweder Indizes, die angegeben werden wie in Kapitel 3, oder (andere) Variable, die angegeben werden mit for all ...

Sowohl bei sequentieller wie auch bei paralleler Ausführung soll, bei vorgegebener Parametermenge zur Beschreibung der Mehrfachoperationen, die Ausführung bedingt ausgewählt werden können durch Bedingungen abhängig vom Parameterwert oder allgemeiner: einer Parametereigenschaft. Dem entspricht eine Berücksichtigung von Auswahl- und Relevanzmaske. Die Sprachelemente sind: when für sequentielle Mehrfachoperationen und where für parallele Mehrfachoperationen. Sie stehen hinter der Parameterangabe und werden gefolgt von einer Bedingung. Die Bedingung kann zusammengesetzt sein, dann steht sie in Klammern; die Teilbedingungen werden mit and zusammengefügt.

2. Auf Benutzerniveau kommt man nicht mehr mit impliziten Deklarationen aus. Es müssen nicht nur verschiedenartige einfache und komplizierter zusammengesetzte Datentypen berücksichtigt werden, sondern es muß insbesondere ein Mechanismus bereitgestellt werden, der es erlaubt, mit dem Wert (oder den Eigenschaften) eines Teildatums das Gesamtdatum oder einen anderen Teil davon aufzurufen.

In /40/ gibt U. Pape eine Übersicht über heute übliche Datenstrukturen für die Programmierung von Graphenalgorithmen und zeigt die Abhängigkeit der Effizienz von den Datenstrukturen und den auf ihnen auszuführenden Operationen auf. Alle diese Datenstrukturen - verkettete Listen, hash-codierte Listen und binäre Bäume - sind konzipiert für sequentielle Algorithmen, bei denen es entweder auf einfache Vergrößerung oder Verkleinerung von (ungeordneten) Mengen ankommt oder auf raschen Zugriff auf Elemente mit kleinstem Gewicht; oft müssen auch Operationen von beiden Typen in ein und demselben Algorithmus ausgeführt werden, was einen gewissen Aufwand für Umsortieren kostet. Für die parallelen und assoziativen Algorithmen hier sparen wir uns dieses Umsortieren auch dort, wo sonst nicht eigentlich parallel gearbeitet werden kann.

Zur Definition (Deklaration) von Datenstrukturen dient def ... end def. Auf def folgen, getrennt durch Semikolons, die Deklarationen der einzelnen Datenbestände. Es gibt zwei Typen von Datenbeständen: Dateien (das sind ungeordnete Mengen von Datenelementen) und reguläre Felder. Datei-Deklarationen beginnen mit file und enthalten eine Liste aus: der primären Datenmenge und, soweit vorhanden, den zugeordneten Hilfsdaten, dargestellt als Funktionen

der primären Daten - die Liste steht in Klammern, die Elemente der Liste sind durch Kommata getrennt - , gefolgt, soweit vorhanden, durch eine Beschreibung der Hilfsfunktionen. Die Deklarationen regulärer Felder beginnen mit field gefolgt, in Klammer, von der Liste der Anzahlen der Elemente in den einzelnen Dimensionalitäten, getrennt durch Kommata; dann folgt die Liste der Hilfsfunktionen, wobei das cartesische Produkt der Indexmengen implizit als primäre Datenmenge zugrunde gelegt wird - diese Liste steht ebenfalls in Klammern, die Elemente der Liste sind getrennt durch Kommata. Die Beschreibung aller Hilfsfunktionen je zu einem Datenbestand wird begrenzt durch by ... end by; die einzelnen Teilbeschreibungen sind getrennt durch Semikolons. Die Beschreibungen enthalten, wie in der Mathematik üblich, Funktionsnamen, Definitionsbereich und Wertebereich. Alle Funktionen, soweit sie nicht ihrer Natur nach primäre Daten sind - etwa: Adjazenzmatrix, Gewichtematrix - werden initialisiert; wo es sich um ausgesprochene Hilfsfunktionen handelt, deren Anfangswert nicht gebraucht wird, schreiben wir undefined, sonst werden die Werte explizit angegeben. Ist eine Initialisierung unabhängig von Argumenten der Funktion, schreiben wir statt dieser Argumente einen Punkt. Die primären Daten der Dateien - etwa: Knotenmenge, Kantenmenge - werden nicht formal deklariert.

Die Definitionen enthalten explizit keine Angaben über die maschineninterne Darstellung der Daten. Sie eignen sich aber trotzdem zur Abschätzung des Speicherbedarfs, indem man Abschätzungen über den Speicherbedarf für die Werte der Hilfsfunktionen benutzt.

3. Hat eine Funktion als Wertebereich ein cartesisches Produkt, so werden die entsprechenden Projektionen durch Indizes dargestellt.
4. Operationen werden im allgemeinen mit den in der Mathematik üblichen Symbolen bezeichnet.

Für die Extremalsuche *max*, *min* gehen wir davon aus, daß in einem assoziativen System ein Zeiger geliefert wird und zwar auf das erste Extremum , das heißt also: ein Indexwert. Deshalb ist hier der Indexparameter angegeben (tiefe stellt) außer der Funktion, deren Extremalwert aufgezeigt werden soll. Die Indexmenge kann auch ein cartesisches Produkt sein.

select soll dazu dienen, aus einer nichtleeren Menge ein beliebiges Element auszuwählen, wobei die Menge Parametermenge einer Mehrfachanweisung ist.

5. empty (bei Abfragen) gibt die leere Menge an, auch hier: eventuell die Parametermenge einer Mehrfachanweisung.

6. Als Konvention werden Mengen durch große Buchstaben bezeichnet; Elemente, Funktionen (bzw. deren Werte), Indizes usw. werden durch kleine Buchstaben bezeichnet. Die Knotenmenge V wird als vorgeordnet betrachtet, Knoten werden oft mit ihrem Index in dieser Ordnung identifiziert.

7. Es wird in der Schreibweise kein Unterschied gemacht zwischen den natürlichen Zahlen $\emptyset$ und 1 und den booleschen Zahlen $\emptyset$ und 1.

4.2.3 Aufspannender Baum eines Graphen

Hier soll eine Gruppe von Graphenalgorithmen beschrieben werden, die sich auf zellularen Arrays rechnen lassen. Und zwar soll sich dieser Abschnitt beschränken auf Algorithmen zur Feststellung, ob ein Graph stark zusammenhängend ist, zur Bestimmung eines aufspannenden Baumes, zur Bestimmung der zugehörigen Fundamentalzylinder und eines maximalen aufspannenden Baumes.

4.2.3.1 Test auf starken Zusammenhang

G sei ein gerichteter ungewichteter Graph. Gefragt ist danach, ob je zwei Knoten in beiden Richtungen miteinander verbunden sind. Die Frage läßt sich auch anders formulieren: Sind von einem beliebig gewählten Knoten aus alle anderen Knoten "vorwärts" und "rückwärts" zu erreichen, das soll heißen: längs der Kantenrichtungen oder entgegen den Kantenrichtungen?

Sei G als (unsymmetrische) Adjazenzmatrix A vorgegeben. Dann läßt sich ein Algorithmus angeben, der Matrizen- und Vektorenkalkül für Boolesche Matrizen benutzt: Gesucht werden zwei n -stellige Vektoren P, Q , deren Komponenten binäre

Werte haben. Beide Vektoren sollen durch iterierte Multiplikationen berechnet werden.

$$P_{i+1} = P_i \cdot A$$

$$Q_{i+1} = A \cdot Q_i$$

mit den Anfangswerten $P_0 = Q_0 = (1, 0, \dots, 0)$

Die Iterationen werden abgebrochen, wenn $P_{i+1} = P_i$ bzw. $Q_{i+1} = Q_i$, d.h. nach maximal n Schritten. Der Graph G ist stark zusammenhängend, wenn alle Komponenten von P und alle Komponenten von Q den Wert 1 haben.

Dieser Algorithmus läßt sich nur dann effektiv implementieren, wenn wir in unserem System, etwa zur Hardware-Unterstützung der Vertikalverarbeitung, einen Mechanismus haben, der uns zur Bit-Matrix A auch deren Transponierte bereitstellt. Die Laufzeit ist dann $O(n^2)$.

Wir können aber auch einen assoziativen Algorithmus angeben, der auf Matrix A arbeitet, ähnlich den Algorithmen für zellulare Arrays. Wir benutzen hier außer A deren Transponierte (für die "umgedrehten" Kantenrichtungen). Ein zelluläres Array zur Lösung dieser Aufgabe ist in 4.2.1 skizziert. Hier soll nur der Teil für die ursprünglichen Kantenrichtungen beschrieben werden:

```
def field (n, n) (a)
  "Adjazenzmatrix"

  by a : {1, 2, ..., n}2 → {0, 1}
    "vorgegebene Funktion"

  end by;

  field (n) (p, l, m)
    "Zusammenhangsmarkierung und Hilfsfunktionen"

  by p : {1, 2, ..., n} → {0, 1} : p{1} = 1, p{2} = 0, ..., p{n} = 0;
    "Vektor zur Markierung der Knoten, die längs der Kanten vom Knoten
    mit Index 1 aus erreichbar sind"

  l : {1, 2, ..., n} → {0, 1} : l(.) = undefined;
    "Hilfsvektor zum Rechnen"
```

$m : \{1, 2, \dots, n\} \rightarrow \{\emptyset, 1\} : m(1) = 1, m(2) = \emptyset, \dots, m(n) = \emptyset$
"Hilfsvektor zum Rechnen"

end by

end def;

sequ $i = 1, 2, \dots, n$

"Iterationsschleife"

par $j = 1, 2, \dots, n$

"alle Knoten"

$\ell(j) := \emptyset$

"löschen Hilfsvektor"

end par;

par $j = 1, 2, \dots, n$ where $p(j) = \emptyset$

"alle noch nicht von 1 aus erreichten Knoten"

sequ $h = 1, 2, \dots, n$ when $m(h) = 1$

"Schleife über die zuletzt erreichten Knoten"

$\ell(j) := \ell(j) \vee a(h, j)$

"feststellen, welche Knoten erreichbar sind von den zuletzt erreichten aus"

end sequ;

$m(j) := (p(j) \vee \ell(j)) \oplus p(j);$

"Markierung der neu erreichten Knoten"

$p(j) := p(j) \vee \ell(j)$

"Markierung aller inzwischen erreichten Knoten"

end par;

if $p(1) = p(2) = \dots = \emptyset(n) = 1$

"wenn alle Knoten erreicht"

then exit

```
else if  $m(1) = m(2) = \dots = m(n) = \emptyset$   
    "wenn kein weiterer Knoten erreicht"  
    then exit  
    else  
    end if  
end if  
end sequ
```

Laufzeit: $O(k)$, wo $k = |E|$ für $G = (V, E)$.

Die Laufzeit ist qualitativ günstiger als die Laufzeit für das zellulare Array (vgl. 4.2.1), die mit $O(n^2)$ angegeben ist. Im Gegensatz zum Array sind wir hier nicht mit n festgelegt.

4.2.3.2 Maximaler (minimaler) aufspannender Baum

Ein aufspannender Baum eines Graphen ist ein Baum aus Kanten des Graphen, der alle Knoten enthält, bei beliebig vorgegebener Wurzel. Ein maximaler (minimaler) aufspannender Baum eines gewichteten Graphen ist ein solcher Baum mit maximaler (minimaler) Kantensumme. G ist als stark zusammenhängend vorausgesetzt.

Sei G ein ungerichteter gewichteter Graph, dessen maximalen aufspannenden Baum wir suchen. G sei vorgegeben durch seine *Gewichtematrix* W ; nicht vorhandene Kanten haben das Gewicht $\emptyset$. Als Ergebnis erhalten wir den maximalen aufspannenden Baum in Form seiner Gewichtematrix T . Ein zellulares Array zur Bestimmung eines (beliebigen) aufspannenden Baumes ist in Abbildung 4.2 angegeben. Es braucht $O(n^2)$ Schritte. Es gibt einen anderen schnelleren Algorithmus, der nur $O(n \cdot \lg \lg n)$ Schritte braucht, sich aber nicht in unserem Konzept nachbilden läßt, weil explizit sequentialisiert werden muß.

def field (n, n) (w, τ)

"Gewichtematrix des Graphen und des Baumes"

by $w : \{1, 2, \dots, n\}^2 \rightarrow N$;

"vorgegebene Gewichtematrix, Gewichte sind natürliche Zahlen"

$\tau : \{1, 2, \dots, n\}^2 \rightarrow N : \tau(.,.) = \emptyset$

"Gewichtematrix des Baumes"

end by;

field (n) (κ)

"Markierungsvektor"

by $\kappa : \{1, 2, \dots, n\} \rightarrow \{\emptyset, 1\} : \kappa(.) = \emptyset$

"Markierung, ob Knoten zum Baum gehört"

end by

end def;

par for all (i, j)

$(i_{\emptyset}, j_{\emptyset}) := \max_{\{i, j\}} w(i, j)$

"Indexpaar des Maximums aller Gewichtswerte"

end par;

$\tau(i_{\emptyset}, j_{\emptyset}) := w(i_{\emptyset}, j_{\emptyset})$;

$\tau(j_{\emptyset}, i_{\emptyset}) := w(j_{\emptyset}, i_{\emptyset})$;

"Übernahme der Kante in den Baum"

$\kappa(i_{\emptyset}) := 1$;

$\kappa(j_{\emptyset}) := 1$;

"Markierung der beiden zugehörigen Knoten, daß im Baum enthalten"

sequ $i = 1, 2, \dots, n-2$

"Iterationsschleife"

par for all (i, j) where ($\kappa_i = 1$ and $\kappa_j = \emptyset$)

$(i_{\emptyset}, j_{\emptyset}) := \max_{\{i, j\}} w(i, j)$

"Indizes des Maximums aller Gewichte von Kanten, die schon mit einem Knoten zum Baum gehören"

end par;

$\tau(i_\emptyset, j_\emptyset) := w(i_\emptyset, j_\emptyset);$

$\tau(j_\emptyset, i_\emptyset) := w(j_\emptyset, i_\emptyset);$

"Übernahme der Kante in den Baum"

$\kappa(j_\emptyset) := 1$

"Markierung des neuen Knotens"

end sequ

Laufzeit: $O(n)$

Die Maximumsuche wird jeweils in vielen Zeilen der Matrix ausgeführt, die "abstrakte" Laufzeitabschätzung berücksichtigt diese Tatsache nicht.

4.2.3.3 Aufspannender Baum und seine Fundamentalzyklen

Die Fundamentalzyklen zu einem aufspannenden Baum sind diejenigen Zyklen des Graphen, die eine Kante außerhalb des Baumes haben und eine oder mehrere Kanten des Baumes enthalten.

Zur Vereinfachung betrachten wir hier einen Graphen ohne Kantengewichte. G sei ungerichtet und stark zusammenhängend. Vorgegeben sei die Adjazenzmatrix A . Als Ergebnis erhalten wir die Adjazenzmatrix T und eine weitere binäre $n \times n$ -Matrix F , in der zu jedem Knoten des Baumes alle möglichen Kanten notiert sind, die zu einem anderen Knoten des Baumes zurückführen und also einen Fundamentalzykel identifizieren.

def field $(n, n) (a, b, f)$

"Adjazenzmatrix des Graphen, Adjazenzmatrix des Baumes und Matrix der Außenkanten der Fundamentalzyklen"

by $a : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\};$

"vorgegebene Adjazenzmatrix"

$b : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\} : b(.,.) = \emptyset;$

"Hilfsfunktion zum Aufbauen der Matrix des Baumes"

$f : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\} : f(.,.) = \emptyset$

"Hilfsfunktion zum Aufbauen der Fundamentalzyklen"

end by;

end def;

sequ $h = 1, 2, \dots, n-1$

"Iterationsschleife"

sequ $i = n, n-1, \dots, \text{when } r(j) = \emptyset$

"alle noch nicht im Baum enthaltenen Knoten"

par $j = 1, 2, \dots, n \text{ where } (r(i) = 1 \text{ and } a(i, j) = 1)$

"alle schon im Baum enthaltenen Knoten"

if not empty

then $j_0 := \max j;$

$t(i, j_0) := a(i, j_0);$

"neuer Knoten bzw. Kante des Baumes"

exit

else

end if

end par

end sequ;

$r(j_0) := 1;$

"gewählten Knoten markieren"

par $j = 1, 2, \dots, n \text{ where } r(j) = 1$

"alle Knoten, die schon im Baum sind"

$\delta(j_0, j) := a(j_0, j)$

"übernehmen alle Kanten aus Knoten mit Nummer j_0 zu anderen Knoten im Baum: identifizieren Fundamentalzyklen"

end par

end sequ

Laufzeit: $O(k)$, wo k die Anzahl der Kanten des Graphen ist.

Das zellulare Array, das mit einem Baum seine Fundamentalzyklen liefert, braucht $O(\mathbb{Z})$ Iterationsschritte, wo $\mathbb{Z}$ die Zahl der Zyklen ist; der einzelne Iterationsschritt ist mindestens von der Komplexität $O(n)$.

4.2.4 Kürzeste Wege in einem Graphen

In vielen Algorithmen wird die Information gebraucht, ob sich zwischen zwei Knoten eines Graphen ein Weg (d.h. eine Kette von gerichteten bzw. ungerichteten Kanten) finden läßt. Gelegentlich ist gefragt, zwischen welchen Knotenpaaren ein Weg existiert. Sehr oft aber geht es darum, in einem Graphen, dessen Kanten eine Länge (oder einen Preis oder ein Gewicht) haben, den kleinsten (oder billigsten oder minimalen) Weg zwischen zwei Knoten zu finden. Dabei interessiert einmal der Verlauf eines solchen Weges, ein anderes Mal seine Länge, gelegentlich beides. Implizite Voraussetzung aller Algorithmen, die im folgenden betrachtet werden: Die Gewichte sind nicht-negativ.

U. Pape /38/ beschreibt für zwei sequentielle Algorithmen (von Moore und von d'Esopo) detailliert die Datenorganisation mit Hilfe von zyklischen und verketteten Listen und gibt für eine konkrete Implementierung Laufzeitvergleiche. Leider lassen sich diese Laufzeitmessungen nicht soweit "abstrahieren", daß wir sie hier unsererseits zum Vergleich heranziehen können.

Bei D.B. Johnson /26/ dagegen findet sich eine übersichtliche Zusammenstellung heute bekannter (sequentieller) Algorithmen für das Problem kürzester Wege. Der beste davon hat eine Laufzeit von

$$O(\min(n^{2+\frac{1}{p}} + k, n \cdot (n+k) \cdot \log_2 n))$$

für ein festes ganzzahliges $p > 0$ (n ist die Knotenzahl, k die Kantenanzahl). Für "dichte" Graphen ergibt sich daraus eine optimale Laufzeit

$$O(n \cdot k).$$

"Dicht" ist dabei definiert durch die Bedingung

$$k = \Omega(n^{1+\varepsilon})$$

mit festem positiven ε . $\Omega(f(n))$ ist eine Funktion, für die gilt

$$\Omega(f(n)) > c \cdot f(n)$$

für fast alle n bei beliebigem festem positiven c .

Der Matrixalgorithmus in 4.2.4.2 ist in diesem Sinne nicht optimal, weil er nicht mit $O(k)$ läuft, sondern mit $O(n^2)$.

In /42/ gibt U. Pape eine Bibliographie zu "kürzeste Weglängen und Wege in Graphen und Netzwerken" mit einem kurzen Kommentar: Matrixmethoden seien unwirtschaftlich hinsichtlich des Speicherbedarfs und der Rechenzeit, falls man sich nicht wirklich für die Wege zwischen allen Knotenpaaren interessiere.

Alle diese Aussagen wurden für konventionelle (sequentielle) Berechnung gemacht. M. Minsky und S. Papert /36/ stellen einen seriellen Algorithmus zur Berechnung des kürzesten Weges zwischen zwei Knoten einem ähnlichen Algorithmus gegenüber, der "viel" parallele Hardware benützt:

Gegeben sei ein ungerichteter gewichteter Graph G . Nicht vorhandene Kanten werden formal eingefügt mit dem Gewicht ∞ . Ein Start- und ein Endpunkt sind ausgezeichnet; gesucht ist der kürzeste Weg zwischen ihnen. Als Ergebnis erhalten wir die Länge dieses Weges, nicht seinen Verlauf. Wir könnten aber ohne wesentlichen Zusatzaufwand auch die Liste der durchlaufenen Kanten aufschreiben. Bei sequentieller Ausführung hat dieser Algorithmus eine Laufzeit von $O(n^2)$. Mit $n-1$ unabhängigen parallel arbeitenden Prozessoren reduziert sich die Laufzeit nur auf $O(n \cdot \log_2 n)$, weil zur Feststellung der jeweils kürzesten Kante zum Fortsetzen der Kette im "tennis-match"-Verfahren das Minimum von bis zu n Werten gesucht werden muß. In unserem Parallel- und Assoziativkonzept aber ist eine Extremumsuche eine einzige Operation, d.h. wir können die Laufzeit des Algorithmus auf $O(n)$ reduzieren.

Crane /6/ beschreibt für einen Rechner mit assoziativem Speicher einen Algorithmus, den Verlauf (und bei Bedarf auch nachträglich die Länge) des kürzesten Weges zwischen zwei ausgezeichneten Knoten zu finden. G ist dabei ebenfalls ein ungerichteter gewichteter Graph. Diesen Algorithmus können wir nachbilden und kommen dann auf dieselbe qualitative Laufzeitabschätzung: $O(n)$. Im wesentlichen haben wir dabei die assoziative Version des vorigen Algorithmus mit dem Unterschied, daß wir nicht mit der Summe des schon durchlaufenen Weges und der möglichen nächsten Kante arbeiten, sondern mit der Differenz der nächsten Kante und des Minimums aller möglichen nächsten Kanten.

4.2.4.1 Alle Wege in einem Graphen

Sei G ein gerichteter (ungewichteter) Graph. Er sei gegeben als Adjazenzmatrix A. Gesucht ist die Matrix P aller möglichen Wege (Pfade) in G, d.h. eine Angabe, ob zwischen je zwei Knoten ein Kantenzug existiert.

Die Wegematrix ist die *transitive Hülle* der Adjazenzmatrix und läßt sich einfach berechnen als

$$P = A^{2^{\lceil \log_2 n \rceil}}$$

Die Matrixmultiplikation machen wir mit Booleschen Verknüpfungen, d.h. statt Multiplikation haben wir "Und", statt Addition "Oder". Man kann dabei entweder so vorgehen, daß man die erste Zeile der ersten Operandenmatrix mit der j-ten Spalte der zweiten Operandenmatrix verknüpft und mit der Abfrage, ob dies Ergebnis in allen Stellen 0 ist oder nicht, sofort das i,j-te Bit der Ergebnismatrix hat, das man dann in seiner Zeile bzw. Spalte richtig unterbringen muß.

Oder aber man arbeitet wie bei Matrizen aus Binärzahlen, d.h. verknüpft parallel die Zeilen der ersten Operandenmatrix nacheinander mit dem entsprechenden Element der k-ten Spalte der Zweiten Operandenmatrix und addiert ("Oder") jedesmal. Das Ergebnis ist die ganze Spalte j der Ergebnismatrix.

Die Laufzeit ist für beide Verfahren $O(n^2)$ für eine Matrixmultiplikation, insgesamt also $O(n^2 \cdot \lceil \log_2 n \rceil)$.

Wir setzen wieder stillschweigend einen Mechanismus voraus, der uns jederzeit erlaubt, zu einer Bitmatrix auch deren Transponierte zu erzeugen. Solche Mechanismen hätten wir mit der Konvertierungshilfe, wie sie die Vertikalverarbeitung braucht, also etwa der "Drehscheibe" oder dem Orthogonalspeicher. Müßte man die Transposition (mikro-)programmieren, kostete sie eine Laufzeit von $O(n^2)$, die sich aber zur Multiplikationszeit hinzuaddierte. Die Größenordnung der Gesamtlaufzeit würde sich also nicht ändern.

Im folgenden ein zweiter Algorithmus: Gegeben sei wieder A, gesucht P. P wird durch Iteration berechnet, Anfangswert: $P=A$. *Warshall's Algorithmus*:

```
def field (n,n) (p)
  "Wegematrix"
  by  $p : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\}$ 
    "vorgegeben mit den Werten der Adjazenzmatrix"
  end by
end def;

sequ  $i = 1, 2, \dots, n$ 
  "Iterationsschleife"
  par  $j = 1, 2, \dots, n$  where  $p(i,j) = 1$ 
    "alle Knoten mit Weg von Knoten i aus"
    sequ  $h = 1, 2, \dots, n$  when  $p(h,i) = 1$ 
      "alle Knoten mit Weg nach Knoten i"
       $p(h,j) := 1$ 
      "Weg durch Knoten i"
    end sequ
  end par
end sequ

Laufzeit:  $O(n^2)$ 
```

Im zellularen Array braucht man, unter Berücksichtigung der Signallaufzeiten, ebenfalls $O(n^2)$.

4.2.4.2 Alle kürzesten Wege in einem Graphen

Hier soll ein Algorithmus, der in /27/ für zellulare Arrays beschrieben ist, "aufgebohrt" werden: Dort werden die Längen aller kürzesten Wege in einem gerichteten gewichteten Graphen G berechnet nach einem Algorithmus von Hu /21/; wir wollen außerdem die Wege selbst in ihrem Verlauf notieren.

G sei gegeben als Gewichtematrix W mit " ∞ " für die nicht enthaltenen Kanten. Außerdem benutzen wir die Adjazenzmatrix A, um P (die Wegematrix) zu berechnen. Die kürzesten Wege (Abstände, Distanzen) berechnen wir in einer Matrix D, die außerdem in jedem Matrixelement noch "lexikalische Vektoren" enthält zur Codierung der zugehörigen Kantenzüge. Die Abstände $d_{i,j}$ sind initialisiert mit $w_{i,j}$. Ein "lexikalischer Vektor" soll die Codierung der Knoten in der Reihenfolge sein, wie sie vom Kantenzug durchlaufen werden. Er enthält zu jedem Element der Matrix D n Eintragungen, je eine für jeden Knoten, jede Eintragung ist n+1 Bits lang: Das Bit mit der Nummer h ($h=0,1,\dots,n$) wird markiert, wenn die Knoten eines Teilweges im Iterationsschritt h in den Kantenzug übernommen werden. Bei Initialisierung ist für diejenigen Elemente, die einer Kante (v_i, v_j) mit endlicher Länge entsprechen, die Eintragung zum Knoten i: $(1,0,\dots,0)$, entsprechend dem Iterationsschritt 0, die Eintragung zum Knoten j: $(0,1,0,\dots,0)$, entsprechend dem Iterationsschritt 1. Zur Vereinfachung haben wir eine n-stellige Codierung, in der nur notiert wird, daß ein Knoten im Kantenzug vorkommt.

def field (n,n) (p,d,l,m)

"Wegematrix, Distanzmatrix mit lexikalischem Vektor und Inzidenzmarkierung"

by p : $\{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\}$;

"vorgegeben mit Adjazenzmatrix"

d : $\{1, 2, \dots, n\}^2 \rightarrow N \cup \{\infty\}$;

"vorgegeben mit Gewichtematrix, Gewichte seien natürliche Zahlen"

$l : \{1, 2, \dots, n\}^3 \rightarrow \{\emptyset, 1\}^{n+1} : l(i, j, k) = \begin{cases} (p(i,j), 0, 0, \dots, 0), & \text{wenn } k = i; \\ (\emptyset, p(i,j), 0, \dots, 0), & \text{wenn } k = j; \\ (\emptyset, 0, \dots, 0) & \text{sonst} \end{cases}$
 "lexikalischer Vektor"

m : $\{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\}^n : m(i, j) = \emptyset$

"Inzidenzmarkierung"

end by;

field (n) (e)

"Hilfsfeld zum Rechnen"

by e : $\{1, 2, \dots, n\} \rightarrow N$; e(.) = undefined

end by

end def;

sequ $h = 1, 2, \dots, n$

"Iterationsschleife, Index des Knotens, bei dem angesetzt wird"

sequ $i = 1, 2, \dots, n$ when $p(i, h) = 1$

"alle Knoten mit Weg zu Knoten h"

par $j = 1, 2, \dots, n$ where $p(h, j) = 1$

"alle Knoten mit Weg von Knoten h"

$p(i, j) := 1;$

"zusammensetzen Weg"

$e(j) := d(i, h) + d(h, j)$

"berechnen zugehörige Wegelänge"

end par;

par $j = 1, 2, \dots, n$ where $(p(h, j) = 1 \text{ and } e(j) < d(i, j))$

"alle Knoten, wo beim Zusammensetzen kürzerer Weg entstanden ist"

$d(i, j) := e(j);$

"übernehmen neue Distanzen"

sequ $g = 1, 2, \dots, n$ when $m_g(h, j) = 1$

"alle Knoten auf angekettetem Teilweg"

$m_g(i, j) := m_g(h, j);$

"kopieren Inzidenzmarkierung dieser Knoten für den neuen Weg"

$\ell(i, j, g) := \ell(h, j, g);$

"kopieren entsprechenden Anteil des lexikalischen Vektors"

$\ell_h(i, j, g) := 1$

"markieren, daß in Schritt h kopiert wurde, d.h. daß entsprechende Knoten in Schritt h angesetzt werden"

end sequ

end par

end sequ

end sequ

Laufzeit: $O(n^3)$; ohne Aufzeichnung des Wegeverlaufs brauchen wir $O(n^2)$

Speicherplatz für die Daten: n^2

$+ n^2 \cdot (b + n(n+1) + n)$

$+ n \cdot b$

Bits.

wobei b die Anzahl der Bits ist, die zur Darstellung der Gewichte (Distanzen) benötigt werden. Ohne Aufzeichnung des Wegeverlaufs brauchen wir $n^2 + n^2 \cdot b + n \cdot b$ Bits.

Die Aufzeichnungen des Wegeverlaufs müssen für die Weiterverwendung unter Umständen umcodiert werden, etwa in die Sequenz der durchlaufenen Knoten bzw. deren Nummern: Diese Reihenfolge wird angegeben durch die lexikalische Anordnung der "lexikalischen Vektorkomponenten". Würde man die Kantenzüge unmittelbar als Knotenfolge aufschreiben, müßte man sie einzeln sequentiell kopieren beim Zusammensetzen. Die Gesamtlaufzeit wäre dann $O(n^4)$.

Die Möglichkeit, außer den Distanzen mit beliebiger Zahlengenauigkeit die Wegeverläufe zu berechnen, geht über die Möglichkeiten der zellularen Arrays hinaus.

4.2.5 Eliminationsgraphen

J.R. Bunch /3/ erwähnt die Beziehung zwischen der Gauss-Elimination von dünnen Matrizen und Eliminationsgraphen: Man sucht eine (neue) Numerierung der Variablen bzw. eine Reihenfolge bei der Elimination, die möglichst keine nichttrivialen Elemente erzeugt an Stellen, wo schon Null steht, höchstens zufällig weitere Nullen. Eine solche Reihenfolge wird sich nicht immer finden lassen.

Sei gegeben eine dünne $n \times n$ -Matrix M . Ihr zugeordnet wird der Graph $G = (V, E)$ mit der Knotenmenge $V = \{v_1, v_2, \dots, v_n\}$ und der Kantenmenge $E: (v_i, v_j) \in E$ genau dann, wenn für das Matrixelement m_{ij} gilt: $m_{ij} \neq 0$ für $i \neq j$. Der Gauss-Elimination von M entsprechend wird folgender Prozeß für G definiert: Eliminieren wir einen Knoten v_h aus G , so löschen wir mit v_h alle Kanten, die zu ihm führen, also (v_i, v_h) ; als neue Kanten führen wir ein: (v_i, v_j) mit dem eben erwähnten i und allen j so, daß $(v_h, v_j) \in E$ war; d.h. wir verbinden alle Knoten, die Kanten zum eliminierten Knoten hatten. Im allgemeinen vermehren wir dadurch die Kanten. Führen wir den beschriebenen Prozeß nacheinander für alle Knoten $v \in V$ durch (indem wir also für jeden folgenden Schritt den veränderten Graphen voraussetzen), so hängt das Ergebnis von der vorgegebenen Ordnung der Knoten $v \in V$ ab. Der Graph G' , der aus dem ursprünglichen Graphen G und den zusätzlich *eingefüllten* Kanten

besteht, heißt *Eliminationsgraph* /48/ Eine Ordnung, bei der zum Schluß die Menge der eingefüllten Kanten leer ist, heißt *perfekte Eliminationsordnung*, der zugehörige Graph $G'=G$ heißt *perfekter Eliminationsgraph*. Eine Ordnung mit einer restlichen Kantenmenge, die in der für jede andere Ordnung echt enthalten ist, heißt *minimal*. Sie heißt *Minimum*, wenn ihre restliche Kantenmenge weniger Elemente hat als alle anderen. Zur Vereinfachung setzen wir M als symmetrisch voraus.

In /48/ werden eine minimale Ordnung für einen beliebigen Graphen und eine perfekte Ordnung für einen beliebigen perfekten Eliminationsgraphen gesucht; außerdem wird getestet, ob ein Graph ein perfekter Eliminationsgraph ist. Alle beschriebenen Algorithmen sind streng sequentiell. Ich habe zunächst diese Algorithmen assoziativ "beschleunigt"; es erscheint aber besser, sie durch andere Algorithmen zu ersetzen, die näher am Ausgangsproblem (der Gauss-Elimination von dünnen Matrizen) bleiben, indem sie mit der Matrixdarstellung des Graphen arbeiten. Das "Auffüllen" ist ein algebraischer Algorithmus, würde sich also zur Implementierung durch ein zelluläres Array anbieten. Die beiden anderen Algorithmen aber sind enumerativ, sie sind vom Typ "depth-first-search". Ihre Implementierung durch ein zelluläres Array würde, ähnlich dem 2. Beispiel in 4.2.1, eine kräftige Unterstützung durch zusätzliche Rechenkapazität außerhalb des Arrays erfordern.

Es ist für einen der drei Algorithmen gelungen, eine qualitativ bessere abstrakte Laufzeit zu erreichen als für den entsprechenden sequentiellen Algorithmus.

4.2.5.1 Auffüllen eines Graphen

Gegeben sei ein Graph $G = (V, E)$ mit ungerichteten Kanten. Es sollen in beliebig vorgegebener Reihenfolge die Knoten eliminiert werden. In /48/ ist ein sequentieller Algorithmus angegeben: es werden Nachfolgerlisten angelegt, das sind zu jedem Knoten die Liste aller Knoten, die mit ihm verbunden sind und eine höhere Nummer haben als der Knoten selbst. Es muß dafür gesorgt werden, daß beim Auffüllen keine Duplikate von schon vorhandenen Eintragungen entstehen. Der angegebene Algorithmus hat - ohne Anlegen der Nachfolgerlisten - eine Laufzeit von $O(n+k')$, wo $n=|V|$ und $k'=|E'|$; E' ist die Kantenmenge des Eliminationsgraphen G' .

Prüfen wir, ob wir den Algorithmus "beschleunigen" können mit Hilfe von assoziativen und parallelen Operationen. Zugrundeliegende Datenmengen sind V (mit Numerierungsfunktion $f : V \rightarrow \{1, 2, \dots, n\}$) und E . Die ungerichteten Kanten werden dargestellt durch je beide zugeordnete gerichtete Kanten, die als geordnete Knotenpaare dargestellt werden. Die Kantenmenge E' des Eliminationsgraphen wird initialisiert mit $E'=E$. V und f sind nicht unter die expliziten Deklarationen aufgenommen worden, weil davon auszugehen ist, daß die Elemente aus V unmittelbar mit ihrer vorgegebenen Nummer codiert werden für die Darstellung der Elemente von E .

def file (E', g)

"Kanten des Eliminationsgraphen mit Markierungsbit"

by $g : E' \rightarrow \{\emptyset, 1\} : g(.,.) = 1$

"Markierungsbit, ob Kante noch nicht ausgewertet wurde, initialisiert mit 1"

end by

end def

sequ $k = 1, 2, \dots, n-1$

$v = f^{-1}\{k\};$

"Schleife über die Knoten in der vorgegebenen Reihenfolge ihrer Elimination"

par for all w where $\{(v, w) \in E' \text{ and } g(v, w) = 1\}$

$g(v, w) := \emptyset$

"löschen Markierungsbit"

$\delta_0 := \min_w \delta(w);$

"suchen kleinsten Index der Knoten, die mit v verbunden sind"

$u := f^{-1}\{\delta_0\}$

end par;

sequ for all w when $\{(w, v) \in E' \text{ and } g(w, v) = 1 \text{ and } w \neq u\}$

"Schleife über alle Knoten, die mit v verbunden sind, außer dem mit kleinstem Index"

$g(w, v) := \emptyset;$

"löschen Markierungsbit"

if $(u, w) \in E'$

"prüfen, ob Kante schon vorhanden"

then $E' := E' \cup \{(u,w), (w,u)\};$

$g(u,w) := 1;$

$g(w,u) := 1$

"aufnehmen neue Kante, markieren als noch nicht ausgewertet"

else

end if

end sequ

end sequ

Wenn die Abfrage $(u,w) \in E'$ immer negativ beantwortet wird, ist G ein perfekter Eliminationsgraph, und die benutzte Ordnung ist eine perfekte Eliminationsordnung.

Laufzeit: $O(n+k')$,

Speicherplatz für die Daten: k' Eintragungen $\hat{=} 4 \cdot \lceil \log_2 n \rceil + 2$ Bits.

Ein Zeitgewinn gegenüber dem sequentiellen Algorithmus /48/ tritt ausschließlich dadurch auf, daß keine geordneten Nachfolgerlisten verwaltet zu werden brauchen. Bei dem vorliegenden Problem, wo diese Listen unmittelbar, in geordneter Form, aus den Matrizen erzeugt werden können, ist der Zeitgewinn gering.

Ich schlage stattdessen vor, die Daten in Matrixform zu belassen. Wir "vergrößern" die Ausgangsmatrix M zur Adjazenzmatrix A :

$$a_{ij} = \emptyset \quad \Leftrightarrow \quad m_{ij} = \emptyset$$

$$a_{ij} = 1 \quad \Leftrightarrow \quad m_{ij} \neq \emptyset$$

Das heißt für den zugehörigen Graphen $G=(V,E)$:

$$a_{ij} = 1 \quad \Leftrightarrow \quad (v_i, v_j) \in E$$

Wir führen auf der Adjazenzmatrix A eine schematische Gauss-Elimination durch:

def field (n,n) (a)

"Adjazenzmatrix"

by $a : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\}$

"vorgegeben"

end by;

field (n) (m)

"Auswahlmaske zum Markieren, ob noch nicht eliminiert"

by $m : \{1, 2, \dots, n\} \rightarrow \{\emptyset, 1\} : m(.) = 1$

end by

end def;

sequ $h = 1, 2, \dots, n-1$

"Schleife über die Zeilen"

par $j = 1, 2, \dots, n$ where $(a[j, h] = 1 \text{ and } m[j] = 1)$

$j_0 := \min_j j$

"kleinster Index für ein nichttriviales Matricelement in Zeile h"

end par;

sequ $i = 1, 2, \dots, n$ when $(a[i, j_0] = 1 \text{ and } m[i] = 1)$

"Schleife über die Zeilen, in denen noch gearbeitet wird und wo ebenfalls das Element mit Index j_0 vorkommt"

par $j = 1, 2, \dots, n$ where $m[j] = 1$

$a_1[i, j] := a_1[i, j] \vee a[h, j]$

"auffüllen von Zeile i mit Zeile h"

end par

end sequ

$m[j_0] := \emptyset$

"eliminieren Element mit Index j_0 "

end sequ

Laufzeit: $O(n+k')$;

Speicherplatz für die Daten: $n^2 + n$ Bits

Die beiden Algorithmen haben im wesentlichen dieselbe Kontrollstruktur. Aber die einzelnen Operationen sind weniger komplex für den Matrixalgorithmus.

4.2.5.2 Perfekte Ordnung eines perfekten Eliminationsgraphen

In /48/ ist ein Algorithmus angegeben, mit dem man für einen perfekten Eliminationsgraphen eine perfekte Ordnung seiner Knoten finden kann; ist der Graph aber nicht perfekter Eliminationsgraph, ist die konstruierte Ordnung nicht einmal minimal. Der zitierte Algorithmus geht von einer symmetrischen Matrix M aus, d.h. er arbeitet auf einem ungerichteten Graphen G . Die Daten werden als verkettete Listen bereitgestellt. Die Laufzeit ist $O(n+k)$ mit $n=|V|$, $k=|E|$. Dieselbe Abschätzung wird angegeben für den Speicherbedarf der Daten.

Ich beschreibe wieder zunächst die assoziativ "beschleunigte" Variante des zitierten Algorithmus, dann den Algorithmus für matrizenförmige Daten.

Auch hier stellen wir die ungerichteten Kanten dar durch zwei geordnete Knotenpaare. Die neuen Knotennummern werden vergeben nach der Größe der Label, d.h. nach der Größe der neuen Nummern ihrer Nachbarn. Das bedeutet: Zeilen (oder Spalten) der Matrix M , die beim Eliminieren möglicherweise verändert worden sind, sollen ihrerseits möglichst bald eliminiert werden.

def file (E, g)

"Kanten mit Löschbit"

by $g : E \rightarrow \{\emptyset, 1\} : g(.,.) = 1$

"Markierung, ob Kante noch nicht gelöscht, initialisiert mit 1"

end by;

file (V, e, h, ℓ)

"Knoten mit neuer Numerierung, Markierungsbit und Label"

by $e : V \rightarrow \{1, 2, \dots, n\} : e(.) = \emptyset;$

"neue Numerierung der Knoten"

$h : V \rightarrow \{\emptyset, 1\} : h(.) = \emptyset;$

"Markierung, ob neue Nummer zugeordnet"

$\ell : V \rightarrow \{\emptyset, 1\}^n : \ell(.) = (\emptyset, \emptyset, \dots, \emptyset)$

"Label zur Notierung, welche Werte von neuen Nummern Knoten haben, die mit dem betrachteten Knoten verbunden sind"

end by

end def;

sequ $i = n, n-1, \dots, 1$

"Schleife über die neuen Knotennummern"

par for all v where $h(v) = \emptyset$

"für alle noch nicht neu nummerierten Knoten"

$\delta_\emptyset := \max_v \ell(v);$

"suche Index des Knotens mit dem größten Label"

$u := \delta^{-1}(\delta_\emptyset);$

$e(u) := i;$

$h(u) := 1$

"numriere Knoten mit dem größten Label mit i "

end par

sequ for all v when $(h(v) = \emptyset \text{ and } (v, u) \in E)$

$g(u, v) := \emptyset;$

$g(v, u) := \emptyset;$

"löschen Kante als ausgewertet"

$\ell_{n-i+1}(v) := 1$

"setze das i -Bit im Label von v auf 1, Numerierung der Labelbits läuft rückwärts wegen Maximumsuche"

end sequ

end sequ

Laufzeit: $O(n+k);$

Speicherplatz für die Daten: $k \cdot (4 \cdot \lceil \log_2 n \rceil + 2)$
 $+ n \cdot (2 \cdot \lceil \log_2 n \rceil + 1 + n)$ Bits

Auch hier lassen sich nur die Suchvorgänge parallelisieren, das heißt: Man kann sich das Vorsortieren der Daten sparen und das Verwalten vereinfachen. Für das Aufdatieren der Daten muß sequentiaisiert werden, weil - auch bei Zusammenfassung der beiden Datenstrukturen - mehrere Zugriffsschlüssel auftreten. Gerade diese Einschränkung - Suchen nach je genau einem Schlüssel - bedeutet eine Begrenzung der Möglichkeiten *aller assoziativ arbeitenden Systeme*.

Der folgende Algorithmus arbeitet wieder mit der Adjazenzmatrix. Er liefert als Ergebnis die neue Numerierung der Knoten in Form einer Permutationsmatrix, wo die Spalten der ursprünglichen Numerierung entsprechen und die Zeilen der neuen (der perfekten Ordnung).

def field (n,n) {a,p}

"Adjazenzmatrix mit Permutationsmatrix"

by $a : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\};$

"vorgegeben"

$p : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\} : p(.,.) = \emptyset$

"Hilfsfunktion zum Aufbauen der Permutationsmatrix"

end by;

field (n) (l,m)

"Label und Auswahlmaske"

by $l : \{1, 2, \dots, n\} \rightarrow \{\emptyset, 1\}^n : l(.) = \{\emptyset, \emptyset, \dots, \emptyset\};$

"Label zum Notieren der neuen Nummern der Nachbarknoten"

$m : \{1, 2, \dots, n\} \rightarrow \{\emptyset, 1\} : m(.) = 1$

"Maske zum markieren, ob noch nicht neu numeriert"

end by

end def;

sequ $i = n, n-1, \dots, 1$

"Schleife über die neuen Knotennummern"

par $j = 1, 2, \dots, n$ where $m(j) = 1$

"alle noch nicht neu numerierten Knoten"

$j_0 := \max_j l(j)$

"Index des Maximums der Label"

end par;

$p(i, j_0) := 1;$

" j_0 wird die neue Nummer i zugeordnet"

$m(j_0) := \emptyset;$

"markieren als neu numeriert"

par $j = 1, 2, \dots, n$ where $(a(j_0, j) = 1 \text{ and } m(j) = 1)$

"alle Nachbarknoten, die noch keine neue Nummer haben"

$l_{n-i-1}(j) := 1$

"einschreiben in Label Markierungsbit, daß zugehöriger Knoten den Nachbarknoten mit der neuen Nummer i hat; Numerierung der Labelbits läuft rückwärts, wegen Maximumsuche"

end par

end sequ

Laufzeit: $O(n)$;

Speicherplatz für die Daten: $3n^2 + n$ Bits

Durch die Matrixdarstellung der Daten haben wir einen erheblichen qualitativen Zeitgewinn.

4.2.5.3 Minimale Eliminationsordnung eines Graphen

Mit dem folgenden Algorithmus, der ebenfalls in /48/ angegeben ist, können wir eine minimale Ordnung auf einem Graphen G finden. Zugleich wird der zugehörige Eliminationsgraph, genauer: seine Kantenmenge E' bestimmt.

Der sequentielle Algorithmus arbeitet, wie der in 4.2.5.2 mit Labeln zur lexikalischen Vorsortierung der noch nicht neu nummerierten Knoten. Die Suchvorgänge gehen über mehrere Schritte: Man sucht nicht nur unmittelbare Nachbarn (zum Labeln), sondern alle überhaupt erreichbaren Knoten, wobei alle, auch die Zwischenknoten, noch unnummeriert sein sollen, und wobei die Zwischenknoten kleinere Label als der Endknoten einer Kette haben sollen. Der in /39/ angegebenen Algorithmus braucht $O(n \cdot k)$ Laufzeit und $O(n+k)$ Speicherplatz, wobei wieder $n=|V|$, $k=|E|$ für $G=(V,E)$ und $k'=|E'|$ für den Eliminationsgraphen $G'=(V,E')$. Auch hier wird M und damit G als symmetrisch vorausgesetzt.

Bei der assoziativen Variante des zitierten Algorithmus benutze ich wieder dieselbe Darstellung von E bzw. E' wie in den vorigen Abschnitten: Jede ungerichtete Kante wird dargestellt durch die beiden zugeordneten gerichteten Kanten, d.h. geordneten Punktepaare.

def file (V, e, f, g, h, n, s)

"Knoten mit neuer Numerierung und Markierung, Label mit Hilfsbit, Hilfsfeldern"

by $e : V \rightarrow \{1, 2, \dots, n\} : e(\cdot) = \emptyset$;

"neue Numerierung der Knoten, initialisiert mit $\emptyset$ "

$f : V \rightarrow \{\emptyset, 1\} : f(\cdot) = \emptyset$;

"Markierung, daß neu nummeriert"

$g : V \rightarrow \{\emptyset, 1\}^n : g(\cdot) = (\emptyset, \emptyset, \dots, \emptyset, 1)$;

"Label, in 1-aus-n-Darstellung"

$h : V \rightarrow \{\emptyset, 1\} : h(\cdot) = \emptyset;$

"Hilfsbit zum Verlängern des Labels"

$\kappa : V \rightarrow \{\emptyset, 1\}^n : \kappa(\cdot) = \text{undefined};$

"Wert des Labels eines Knotens, der von v aus noch erreichbar ist"

$s : V \rightarrow \{\emptyset, 1\} : s(\cdot) = \text{undefined}$

"Markierungsbit, ob in einer Kette erreicht"

end by;

file (E);

"ursprüngliche Kantenmenge, vorgegeben"

field (E');

"Kantenmenge des Eliminationsgraphen mit der neuen Numerierung, initialisiert als leer"

field (l) (l)

"Anzahl der verschiedenen Labelwerte, initialisiert mit 1"

end def;

sequ $i = n, n-1, \dots, 1$

"Schleife über die neuen Knotennummern"

par for all v where $\{f(v) = \emptyset \text{ and } g(v) = l\}$

"für alle noch nicht neu nummerierten Knoten mit maximalem Labelwert"

$u := \text{select } (v)$

"wähle einen Knoten"

end par;

$e(u) := i;$

$f(u) := 1;$

"gewählter Knoten bekommt neue Nummer i"

par for all v where $f(v) = \emptyset$

$\kappa(v) := (\emptyset, \emptyset, \dots, \emptyset);$

$s(v) := \emptyset$

"markieren als noch nicht in einer Kette erreicht"

end par;

sequ for all v when $((u,v) \in E \text{ and } f(v) = \emptyset)$

"alle noch nicht nummerierten Nachbarknoten des neu nummerierten"

$r(v) := g(v);$

"höchster Labelwert in der Kette bei Erreichen dieses Knotens"

$s(v) := 1;$

"markieren als erreicht in Kette"

$h(v) := 1;$

"erreichbar in Kette der vorgeschriebenen Form: labeln"

$E' := E' \cup \{(u,v)\}$

"Kante gehört zum neuen Eliminationsgraphen"

end sequ;

sequ $j = 1, 2, \dots, \ell$

"Schleife über die ℓ verschiedenen Labelwerte, der Größe nach"

sequ for all v when $r_j(v) = 1$

"für alle Knoten, die erreichbar sind über Ketten mit Labelwerten bis j "

$r_j(v) := \emptyset;$

sequ for all w when $((v,w) \in E \text{ and } f(w) = \emptyset \text{ and } s(w) = \emptyset)$

"alle Nachbarknoten, die unnummeriert und noch nicht erreicht sind"

$s(w) := 1;$

"markieren als erreicht in Kette"

if $f(w) > j$

"wenn Label noch größeren Wert hat"

then $h(w) := 1;$

"erreichbar in Kette der vorgeschriebenen Form: labeln"

$r(w) := g(w);$

"Kette hat jetzt Label bis zu diesem Wert"

$E' := E' \cup \{(u,w)\}$

"neue Kante des Eliminationsgraphen gefunden"

else $r(w) := 1$

"Kette hat weiterhin Label bis zum Wert j "

end if

end sequ

end sequ

end sequ;

$k := 1;$

"vorbesetzen Zähler zum Sortieren und Auszählen der neuen Labelwerte"

sequ $j = 1, 2, \dots, \ell$

"Schleife über die alten Labelwerte"

par for all v where $\{f(v) = \emptyset \text{ and } g_j(v) = 1 \text{ and } h(v) = \emptyset\}$

"alle noch nicht numerierten Knoten mit dem Labelwert j und ohne Zusatzbit im Label"

if not empty

then $g_j(v) := \emptyset;$

$g_k(v) := 1;$

"alten Labelwert löschen, neuen eintragen"

$k := k+1$

"hochzählen"

else

end if

end par;

par for all v where $\{f(v) = \emptyset \text{ and } g_j(v) = 1 \text{ and } h(v) = 1\}$

"alle noch nicht numerierten Knoten mit dem alten Labelwert j und mit Zusatzbit"

if not empty

then $g_j(v) := \emptyset;$

$g_k(v) := 1;$

"neu labeln"

$k := k+1$

"hochzählen"

else

end if

end par

end sequ;

$\ell := k;$

"neue Zahl verschiedener Labelwerte"

par for all v where $f(v) = \emptyset$

$h(v) := \emptyset$

"löschen Zusatzbit im Label"

end par

end sequ

Laufzeitabschätzungen: Untersuchung der Ketten zum Labeln neuer Knoten: $O(k)$;
Sortieren der Label: $O(n)$; insgesamt also: $O(n \cdot k)$ wie im zugrundeliegenden sequentiellen Algorithmus /48/.

Speicherbedarf für die Daten: $n \cdot (\lceil \log_2 n \rceil + 1 + 2(n+1))$
 $+ k \cdot 4 \cdot \lceil \log_2 n \rceil$
 $+ k' \cdot 2 \cdot \lceil \log_2 n \rceil$
 $+ \lceil \log_2 n \rceil$ Bits.

Bemerkung: Die Werte der Label und der Hilfswerte für die Ketten könnten auch einfach binär codiert werden und bräuchten dann $\lceil \log_2 n \rceil$ Bits jeweils. An dieser Stelle habe ich mich für die 1-aus-n-Codierung entschieden, weil sie mir speziell für die Implementierung des Parallel- und Assoziativkonzeptes durch *Vertikalverarbeitung* am besten geeignet erscheint. In der Zusammenstellung der Abschätzungen am Schluß des Kapitels sind beide Codierungen berücksichtigt.

Auch bei diesem Algorithmus können nur die Zugriffs- und Verwaltungsoperationen assoziativ vereinfacht werden - unabhängig von der Realisierung des assoziativen Zugriffs. Die Ketten zum Beispiel müssen sequentiell aufgebaut werden, weil Daten mit verschiedenen Schlüsseln bearbeitet werden müssen. In einem mikroprogrammierten Parallel- und Assoziativkonzept könnte man Suchoperationen mit mehreren Schlüsseln als Spezialoperationen realisieren. Bei "makroskopischer" Betrachtung würde sich dann die Laufzeit reduzieren auf $O(n \cdot \frac{e}{d})$, wo d die durchschnittliche Verzweigung in den Knoten des Graphen angeben soll.

Es folgt die Umsetzung des Algorithmus auf die Matrixdarstellung des Graphen. Wir gehen wieder aus von der Adjazenzmatrix A . Er liefert als Ergebnis die neue Nummerierung der Knoten in Form einer Permutationsmatrix und die Adjazenzmatrix B des Eliminationsgraphen.

def field (n, n) (a, p, b, c, s, t, u)

"Adjazenzmatrix des Graphen, Permutationsmatrix, Adjazenzmatrix des Eliminationsgraphen und Hilfsfunktionen zum Aufbauen der Ketten beim Labeln"

by $a : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\};$

"Adjazenzmatrix, vorgegeben"

$p : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\} : p(..) = \emptyset;$

"Hilfsfunktion für die Permutationsmatrix"

$b : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\} : b(..) = \emptyset;$

"Hilfsfunktion für die Adjazenzmatrix des Eliminationsgraphen"

$c : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\} : c(.,.) = \text{undefined};$

"Hilfsfunktion zum Kopieren der Adjazenzmatrix für den Aufbau der Ketten"

$s : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\}^n : s(.,.) = \text{undefined};$

"Hilfsfunktion zum Kopieren der Label der Endknoten einer Kante oder eines Kantenzuges"

$t : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\} : t(.,.) = \text{undefined};$

"Hilfsfunktion zum Markieren, daß Endknoten eines Kantenzuges größeres Label hat als die anderen"

$u : \{1, 2, \dots, n\}^2 \rightarrow \{\emptyset, 1\} : u(.,.) = \text{undefined}$

"Hilfsfunktion zum Markieren, daß es sich um einfache Kante handelt, nicht zusammengesetzten Kantenzug"

end by;

field (n) (m, κ)

"Hilfsfunktionen zum Markieren und Labeln der Knoten"

by $m : \{1, 2, \dots, n\} \rightarrow \{\emptyset, 1\} : m(.) = 1;$

"Hilfsfunktion zum Markieren, daß Knoten noch nicht eliminiert"

$\kappa : \{1, 2, \dots, n\} \rightarrow \{\emptyset, 1\}^n : \kappa(.) = (\emptyset, \emptyset, \dots, \emptyset)$

"Label, notiert die neuen Nummern der Nachbarknoten, d.h. deren Eliminationsreihenfolge"

end by

end def;

sequ $i = n, n-1, \dots, 1$

"Schleife über die neuen Knotennummern"

par $j = 1, 2, \dots, n$ where $m(j) = 1$

"alle noch nicht neu nummerierten Knoten"

$j_\emptyset := \max_j \kappa(j)$

"Index des ersten Maximums der Label"

end par;

$p(i, j_\emptyset) := 1;$

" $j_\emptyset$ wird die neue Nummer i zugeordnet"

$\kappa(j_\emptyset) := \emptyset;$

"Label von $j_\emptyset$ wird auf den kleinstmöglichen Wert gesetzt"

sequ $j = 1, 2, \dots, n$ when $m(j) = 1$

par $h = 1, 2, \dots, n$ where $m(h) = 1$

$s(j, h) := \kappa(h);$

"zuordnen allen Kanten Label ihres Endknoten"

$c(j, h) := a(j, h);$

"kopieren Teilbereich der Adjazenzmatrix"

$t(j, h) := \emptyset;$

"löschen Markierungsbit, daß Label für Endknoten am größten"

$u(j, h) := 1$

"setzen Markierungsbit, daß einfache Kante"

end par

end sequ;

par $h = 1, 2, \dots, n$ where $m(h) = 1$

$t(j_0, h) := 1;$

"Knoten, die Kante von j_0 aus haben, erfüllen die Kettenbedingung"

$b(j_0, h) := 1;$

"Kante des Eliminationsgraphen"

$\kappa_i(h) := 1;$

"Labelbit zur neuen Nummer i "

end par;

sequ $j = 1, 2, \dots, n$

"Iterationsschleife für die Berechnung der Ketten"

sequ $h = 1, 2, \dots, n$ when $(m(h) = 1 \text{ and } c(h, j) = 1)$

"alle Teilketten zum Knoten mit Nummer j "

par $\ell = 1, 2, \dots, n$ where $(m(\ell) = 1 \text{ and } c(j, \ell) = 1)$

"alle Teilketten von Knoten mit Nummer j aus"

$c(h, \ell) := 1$

"eintragen Existenz einer Kette"

end par;

par $\ell = 1, 2, \dots, n$ where ($m(\ell) = 1$ and $c(j, \ell) = 1$
and $s(j, \ell) > s(h, j)$)

"alle Ketten von Knoten mit Nummer j aus, bei denen im ange-
 fügten Teil größere Label sind als im früheren"

$s(h, \ell) := s(j, \ell);$

"größtes Label in der Kette"

$t(h, \ell) := t(j, \ell) \vee u(j, \ell)$

"markieren, daß Endknoten größtes Label hat"

end par;

par $\ell = 1, 2, \dots, n$ where ($m(\ell) = 1$ and $c(j_\emptyset, \ell) = 1$
and $t(j_\emptyset, \ell) = 1$)

"alle Knoten mit größtem Label am Ende von Ketten von Knoten
 $j_\emptyset$ aus"

$b(j_\emptyset, h) := 1;$

"Kante im Eliminationsgraph"

$r_i(\ell) := 1$

"Labelbit zur neuen Nummer i "

end par

end sequ

end sequ;

$m(j_\emptyset) := \emptyset$

"löschen Knoten $j_\emptyset$ als bearbeitet, d.h. neu numeriert"

end sequ

Laufzeit: Aufbau der Ketten: $O(n^2)$, insgesamt: $O(n^3)$

Speicherplatz für die Daten: $(4+n) \cdot n^2 + 3n$ Bits.

Wir haben also eine qualitative Laufzeitverschlechterung, weil im allgemeinen
 und speziell bei Graphen, die aus dünnen Matrizen hergeleitet sind $k < n^2$.

4.2.6 Hamilton-Wege in einem Graphen

Hamilton-Wege sind Wege durch einen Graphen, die jeden Knoten genau einmal durch-
 laufen. G ist implizit als zusammenhängend vorausgesetzt.

Die heuristische Motivation, dieses Problem aufzugreifen stammt aus der Beschäftigung mit Algorithmen zur Betriebsmittelverwaltung. V. Mierlea /35/ kommt zum Beispiel auf die Aufgabe, in einem Graphen den kürzesten Hamilton-Weg zu finden.

Auf den ersten Blick sieht es so aus, als könne diese Aufgabe analog zur Bestimmung der Längen und Verläufe der kürzesten Wege (in 4.2.5.2) gelöst werden. Daß diese Vermutung falsch ist, liegt daran, daß beim Suchen nach einem kürzesten Weg sozusagen "lokal" optimiert werden kann, während man bei der Suche nach Hamilton-Wegen oft sehr lange sehr viele Alternativen behalten muß, ehe man die meisten davon eindeutig ausscheiden kann.

Eine Lösung dieses Problems durch zellulare Arrays erscheint mir unmöglich, aber auch bei parallel und assoziativ arbeitenden Systemen treten prinzipielle Schwierigkeiten auf, die nur um den Preis hohen Speicheraufwandes zu lösen sind oder zwingen, das Konzept "aufzuweichen".

4.2.6.1 Alle Hamilton-Wege in einem Graphen

Beim Back-track-Verfahren von Hamilton und Flores /4/ werden die Hamilton-Wege, soweit sie existieren, einzeln ("enumerativ") gefunden. Wählt man als Darstellung für den Graphen G eine Menge von n Listen, in denen für die n Knoten alle Nachfolger notiert sind, so läßt sich das klassische Back-track-Verfahren assoziativ höchstens beschleunigen bei der Abfrage, ob ein neu gewählter Knoten schon zum bisherigen Weg gehört. Laufzeit: $O(n!)$

Das allgemeine Back-track-Verfahren kann beschleunigt werden durch das "Multipiepath"-Verfahren: Die allgemeine Idee hierbei ist, während des Aufbaus eines Suchweges $S_\emptyset$ den Graphen möglichst zu reduzieren - z.B. werden alle Kanten "gestrichen", die im weiteren Verlauf von $S_\emptyset$ garantiert nicht mehr benutzt werden können, nämlich alle weiteren Kanten zu schon durchlaufenen Knoten usw. - , und im modifizierten Graphen werden schon erkennbare längere Kantenzüge zusammengefaßt. Dadurch wird sowohl der Aufbau von $S_\emptyset$ beschleunigt als auch das Erkennen von Abbruchbedingungen.

Der jeweils (in einer Stufe des Back-trackings) aktuelle Zustand des modifizierten Graphen kann durch die zugehörige Stufe eines Kellers von Selektionsmasken fixiert werden. Dieser Keller kann höchstens die Tiefe n haben, es genügt also, festen Platz dafür vorzusehen.

Schwierig ist, Knoten mit Innen- bzw. Außengrad 1 aufzufinden zum Zusammenfassen von Teilkantenzügen. Man muß immer wieder diese beiden Werte sequentiell für alle Knoten ermitteln (z.B. assoziativ aufrufen und Selektionsmaskenbits aufsummieren).

Der Algorithmus für dieses "beschleunigte" Back-track-Verfahren ist so komplex, daß ich darauf verzichte, ihn hier formalisiert aufzuschreiben. Dieser Algorithmus ist im wesentlichen streng sequentiell; es lassen sich aber eine Reihe von Such- und Aufdatierungsoperationen assoziativ (parallel) ausführen. Das bedeutet, daß man sich damit eine raffinierte Datenführung ersparen kann. Laufzeit: $O(n!)$

4.2.6.2 Kürzester Hamilton-Weg in einem Graphen

In /4/ ist ein "algebraisches" Verfahren angegeben, alle Hamilton-Wege eines Graphen zu bestimmen. G ist dort ein gerichteter Graph. Für die Anwendungen zum Beispiel auf Probleme der Betriebsmittelvergabe /35/ sei G außerdem gewichtet. G sei etwa vorgegeben als Gewichtematrix W .

Der zitierte Algorithmus arbeitet iterativ; er sucht in Schritt ℓ alle einfachen Wege (d.h. Wege ohne Knotenwiederholungen) aus genau ℓ Kanten. Das sind maximal $(\ell-1)!$ verschiedene Wege für ein festes Paar von Anfangs- und Endknoten und $n \cdot (n-1) \cdot (\ell-1)!$ Wege insgesamt in Schritt ℓ . Ein Abspeichern der Zwischenergebnisse in Matrixform erscheint nicht mehr sinnvoll.

Formal können wir aber einen Matrizenalgorithmus aufschreiben:

$$H_{i,h}^{(\ell+1)} = \sum_{j=1}^n e_{i,j} \cdot H_{j,h}^{(\ell)}$$

$$v_i \notin H_{j,h}^{(\ell)}$$

Dabei sollen bedeuten: $H_{i,h}(\ell)$: die Menge aller einfachen Wege zwischen den Knoten v_i und v_h aus genau ℓ Kanten; $e_{i,j}$: die Kante $(v_i, v_j) \in E$;
Multiplikation: Konkatenation der Kante mit jedem einzelnen Weg aus der Menge der einfachen Wege zwischen den angegebenen Knoten und von der angegebenen Länge; Summe: Mengenbildung.

Stellen wir jeden Kantenzug dar als geordnete Folge der durchlaufenen Knoten, die etwa durch ihre vorgegebenen Nummern identifiziert werden; ordnen wir ihm zu die Summe der Kantengewichte; und führen wir der Bequemlichkeit halber einen "Inzidenzvektor" mit, der notiert, welche Knoten auf dem Kantenzug liegen (indem wir jedem Knoten ein Bit des binären Vektors zuordnen). Arbeiten wir auf zwei getrennten Speicherbereichen, um dort abwechselnd alle einfachen Wege aus 1 Kante, 2 Kanten, ..., ℓ Kanten, $\ell+1$ Kanten zu erzeugen. Dann läßt sich mit

$$f_{\ell} : H(\ell) \rightarrow N$$

als Gewichtsfunktion (mit $f_1 = w$) und

$$g_{\ell} : H(\ell) \rightarrow \{0, 1\}^n$$

als Funktion zum Markieren der Inzidenz eines Knotens in einem Kantenzug der Matrizenalgorithmus ganz einfach programmieren:

sequ $\ell = 1, 2, \dots, n-1$

"Iteration über die Anzahl der Kanten in einem Weg"

sequ $i = 1, 2, \dots, n$

"Anfangsknoten einer neu hinzuzufügenden Kante"

sequ $j = 1, 2, \dots, n$ when $a(i, j) = 1$

"Endknoten einer neu hinzuzufügenden Kante"

par for all $e^*(j, h)$ where $(e^*(j, h) \in H(\ell) \text{ and } g_{\ell, i}(j, h) = \emptyset)$ $*$)

"alle ansetzbaren Kantenzüge, die den Anfangsknoten der neuen Kante nicht enthalten"

$$H(\ell+1) := H(\ell+1) \cup \{e(i, j) \cdot e^*(j, h)\};$$

"verketteten Kante mit Kantenzug, gibt einfachen Weg der Länge ℓ "

$$g_{\ell+1}(i, h) := g_{\ell}(j, h);$$

$$g_{\ell+1, i}(i, h) := 1;$$

"kopieren Inzidenzvektoren, eintragen Markierung für neuen Anfangsknoten"

$$\delta_{\ell+1}(i, h) := \delta_j(i, j) + \delta_{\ell}(j, h)$$

"addieren Gewichte"

*) $H(\ell)$ sei die Menge aller einfachen Wege aus ℓ Kanten; $e^*(j, h)$ sei ein Kantenzug von v_j nach v_h

```

        end par
    end sequ
end sequ;
if  $H(\ell+1)$  empty
    "wenn es keine einfachen Wege aus  $\ell+1$  Kanten gibt"
then exit
    "abbrechen"
else
end if
end sequ
```

In dieser Schreibweise benötigt der Algorithmus $O(n \cdot k)$ Schritte.

Fügt man die noch fehlenden Datendeklarationen hinzu, zeigt sich, daß der Algorithmus in seiner Implementierung mehr Aufwand kostet, als es zunächst scheinen mag: Entweder haben wir die Daten wirklich matrizenförmig (regulär) gespeichert; dann bedeutet das aber: Wir haben im ℓ -ten Schritt Matrizen der Dimensionalität $\ell+1$, weil jeder einzelne Kantenzug identifizierbar sein muß, und wir müssen nicht "Vektoren" parallel kopieren zum Ansetzen, sondern Matrizen hoher Dimensionalität. Oder aber wir haben Dateien, bei denen wir darauf bedacht sind, die Daten möglichst lückenlos zu speichern, was aber dann ein "stellengetreues" paralleles Kopieren aller ausgewählten Kantenzüge zu vorgegebener neuer Anfangskante unmöglich macht. Bei unserem *mikroprogrammierten* Parallel- und Assoziativkonzept könnten wir uns behelfen entweder, bei Vertikalverarbeitung, mit einem hardware-unterstützten "Pack"-Befehl (vgl. 3.5.2.2) oder, bei konventioneller Emulation, durch einen entsprechenden mikroprogrammierten Befehl.

Ein System mit echtem - das heißt: *hardware-realisiertem* - Assoziativspeicher erlaubt aber die vorgeschlagene Programmtechnik überhaupt nicht, weil es keine Möglichkeit bietet, Datenblöcke als ganzes in andere Speicherbereiche zu kopieren. Hier bliebe als einzige Lösung, will man den Grad an Parallelität erreichen, wie ihn der aufgeschriebene Algorithmus hat, schon bei Initialisierung jede Kante $(n-2)!$ mal zu speichern und jedem Exemplar die Codierung eines anderen "virtuellen" Hamilton-Weges mitzugeben, bei dem diese Kante das letzte Stück des Kantenzuges wäre. Dann könnte man, entsprechend unserem

Algorithmus, Kante um Kante die Wege mit den "realen" Daten fortzusetzen versuchen und alle "virtuellen" Wege ausscheiden, denen die "realen" Daten widersprechen. "Virtuell" soll heißen: Kantenzug des zugehörigen vollständigen Graphen; "real": mit den vorgegebenen Kanten von G . Hier müßten nicht verschiedene Kantenzüge auf einmal kopiert werden, sondern immer nur die eine neu anzusetzende Kante. Die vorgeschlagene Technik besteht also darin, den assoziativen Speicher vorab mit einer Adreßinformation zu füllen, die dann assoziativ ausgewertet werden kann. Hier steckt der Aufwand, neben der Vorbereitung der Daten, - wie beim Matrizen-Algorithmus - in dem enormen Speicherverschnitt, weil wir hier von Anfang an Speicherplatz für $n!$ Kantenzüge beanspruchen. Für die einzelne Eintragung brauchen wir Platz zum Codieren der n Knoten eines Hamilton-Weges, für den Wert des Gewichtes und für zwei Markierungsbits, ob ein Kantenzug real existiert, und zwar für je zwei aufeinanderfolgende Rechenschritte ℓ und $\ell+1$.

Sollen geschlossene Hamiltonwege gesucht werden, genügt es, alle Kantenzüge zu einem festen Knoten zu betrachten, etwa v_1 . Das bedeutet für den angegebenen Algorithmus, daß die parallele Mehrfachanweisung nicht mehr über alle passenden Kantenzüge von v_j zu irgendeinem v_h geht, sondern nur noch über alle passenden Kantenzüge nach v_1 . Als explizite abstrakte Laufzeitverringerung macht sich diese Einschränkung nicht bemerkbar. Konkret bedeutet sie allerdings für das mikroprogrammierte Parallel- und Assoziativkonzept Zeitersparnis und für das hardware-implementierte Assoziativsystem Speicherersparnis (um den Faktor n).

Der kürzeste Hamiltonweg - soweit es überhaupt welche gibt - muß anschließend bestimmt werden.

4.3 Auswertung der Algorithmen für Graphen

Hier sollen die *abstrakten* Abschätzungen für *Zeit-* und *Datenspeicherbedarf* der entwickelten Algorithmen noch einmal zusammengestellt werden, zusammen mit *konkreten* Laufzeitabschätzungen für einzelne Rechnerkonzepte, soweit möglich. Die Rechnerkonzepte sind:

der konventionelle *sequentielle* Rechner;

das mikroprogrammierte Parallel- und Assoziativkonzept mit seinen beiden möglichen Implementierungen (*Vertikalverarbeitung* oder *konventionelle Emulation*);

zellulare Arrays;

assoziative Systeme, die insbesondere erlauben, die Selektorbits zu verschieben, um physikalisch aufeinander folgende Zellen mit logisch aufeinander folgenden Teilen eines langen Datenwortes belegen zu können.

Bei den konkreten Abschätzungen sind für alle Konzepte die Laufzeiten der zugrundeliegenden Einzeloperationen zu berücksichtigen. Beim sequentiellen und beim assoziativen System bedeutet dies: Es müssen die Längen der verarbeiteten Datenworte berücksichtigt werden. Beim mikroprogrammierten Parallel- und Assoziativkonzept muß zusätzlich der Umfang eines "parallel" verarbeiteten Datenbestandes berücksichtigt werden. Alle benutzten Operationen sind übrigens *linear* im Sinne der Definition in 3.2. Im Gegensatz zu den Abschätzungen in Kapitel 3 wird hier die Speicher- und Rechenwerkbreite r als fest angenommen und tritt deshalb in den Abschätzungen nicht auf.

Die Abschätzungen für zellulare Arrays sind selbst für die abstrakte Laufzeit schwer in vergleichbare Form zu bringen, weil in /31/ im allgemeinen nur die Anzahlen von Iterationen angegeben sind, nicht aber explizit die Komplexität der einzelnen Iteration. Auf Angaben zu Datenumfang und konkreten Laufzeiten für zellulare Arrays habe ich ganz verzichtet. Für die Algorithmen der Eliminationsgraphen steht in /48/ keine präzise Speicherplatzabschätzung zu Verfügung, weil die Länge der Einzeldatendarstellung nicht berücksichtigt ist, ich habe die entsprechende Detaillierung nachgetragen.

Bei allen Algorithmen ist der Aufwand zum Vorbereiten der Daten vernachlässigt, also etwa zum Vorsortieren der Daten bei sequentieller Verarbeitung oder zum Laden des impliziten Adressierschemas für den assoziativen Hamiltonalgorithmus.

Es bedeuten in den Abschätzungen in Tabelle 4.1:

- $n = |V|$ die Anzahl der Knoten des vorgegebenen Graphen;
- $k = |E|$ die Anzahl der Kanten des vorgegebenen Graphen;
- $k' = |E'|$ die Anzahl der Kanten bei Elimination;
- Z ist die Anzahl der Fundamentalzyklen;
- b ist die Anzahl der Bits für die Darstellung der Gewichte

Tabelle 4.1: Vergleich der Algorithmen für verschiedene Maschinenkonzepte nach Laufzeitverhalten und Speicheraufwand für die Daten

Problem/Algorithmus	Maschinenkonzept	abstrakte Laufzeit	Datenumfang	konkrete Laufzeit
Test auf starken Zusammenhang /Vektoren-Kalkül /Wegverfolgung	mikroprogrammiert	$O(n^2)$	$O(n^2)$	$O(n^3)$
	mikroprogrammiert	$O(k)$	$O(n^2)$	$O(k \cdot n)$
	zellulares Array	$O(n^2)$	-	-
maximaler aufspannender Baum /"einfacher" Algorithmus	mikroprogrammiert	$O(n)$	$O(n^2 \cdot b)$	$O(n^3 \cdot b)$
	zellulares Array	$O(n^2)$	-	-
	zellulares Array	$O(n \cdot \log_2 n)$	-	-
aufspannender Baum mit Fundamentalzylken /identifizierende Kante /ganze Zyklen	mikroprogrammiert	$O(k)$	$O(n^2)$	$O(k \cdot n)$
	zellulares Array	$O(n \cdot z)$	-	-
alle Wege in einem Graphen /Matrizenkalkül /Marshall's Algorithmus	mikroprogrammiert	$O(n^2 \cdot \log_2 n)$	$O(n^2)$	$O(n^3 \cdot \log_2 n)$
	mikroprogrammiert	$O(n^2)$	$O(n^2)$	$O(n^3)$
	zellulares Array	$O(n^2)$	-	-
kürzeste Wege/mit Verlauf /ohne Verlauf	mikroprogrammiert	$O(n^3)$	$O(n^2 \cdot (b+n^2))$	$O(n^3 \cdot (b+n^2))$
	zellulares Array	$O(n^2)$	-	-

Problem/Algorithmus	Maschinenkonzept	abstrakte Laufzeit	Datenumfang	konkrete Laufzeit
Auffüllen eines Eliminationsgraphen/Datei /Datenmatrix	sequentiell	$O(n+k')$	$O(k' \cdot \log_2 n)$	$O((n+k') \cdot \log_2 n)$
	mikroprogrammiert	$O(n+k')$	$O(k' \cdot \log_2 n)$	$O(n \cdot k' \cdot \log_2 n)$
	assoziativ	$O(n+k')$	$O(k' \cdot \log_2 n)$	$O((n+k') \cdot \log_2 n)$
	mikroprogrammiert	$O(n+k')$	$O(n^2)$	$O((n+k') \cdot n)$
Perfekte Ordnung eines Eliminationsgraphen/Datei /Datenmatrix	sequentiell	$O(n+k)$	$O((n+k) \cdot \log_2 n)$	$O((n+k) \cdot \log_2 n)$
	mikroprogrammiert	$O(n+k)$	$O(n^2 + k \cdot \log_2 n)$	$O(n^3 + k \cdot \log_2 n)$
	assoziativ	$O(n+k)$	$O(n^2 + k \cdot \log_2 n)$	$O(n^2 + k \cdot \log_2 n)$
	mikroprogrammiert	$O(n)$	$O(n^2)$	$O(n^3)$
Minimale Ordnung eines Eliminationsgraphen/Datei	sequentiell	$O(n \cdot k)$	$O((n+k') \cdot \log_2 n)$	$O(n \cdot k \cdot \log_2 n)$
	mikroprogrammiert, vertikal	$O(n \cdot k)$	$O(n^2 + (k+k') \cdot \log_2 n)$	$O(n^2 \cdot k)$
	mikroprogrammiert, konventionell	$O(n \cdot k)$	$O((n+k+k') \cdot \log_2 n)$	$O(n \cdot k \cdot \log_2 n)$
	assoziativ	$O(n \cdot k)$	$O(n^2 + (k+k') \cdot \log_2 n)$	$O(n^2 \cdot k)$
/Datenmatrix	mikroprogrammiert	$O(n^3)$	$O(n^3)$	$O(n^4)$
alle Hamilton-Wege /Back-track-Verfahren	sequentiell	$O(n!)$	$O(n! \cdot n \cdot \log_2 n)$	$O(n! \cdot n \cdot \log_2 n)$
kürzester Hamilton-Weg/ algebraisches Verfahren	mikroprogrammiert, mit "Packen"	$O(n \cdot k)$	$O(n! \cdot (n \cdot \log_2 n + k))$	$O(n \cdot k \cdot n! \cdot (n \cdot \log_2 n + b))$
	assoziativ	$O(n \cdot k)$	$O(n! \cdot (n \cdot \log_2 n + b))$	$O(n \cdot k \cdot (n \cdot \log_2 n + b))$

4.3.1 Zusammenfassung und Interpretation der Vergleiche

Es erscheint mir als das ergiebigste, die Algorithmen, wie sie hier mehr oder weniger zufällig zusammengetragen wurden, zu klassifizieren nach den zugrundeliegenden Datenstrukturen: in Datei-Algorithmen und Feld-Algorithmen.

Unter *Datei* verstehe ich hier eine Menge von Daten, deren Umfang prinzipiell nicht bekannt ist. Die Daten werden durch Eigenschaften, Werte, Kriterien identifiziert. Praktisch bedeutet das: Entweder arbeiten wir auf einer Menge nicht sortierter Daten *assoziativ* oder auf Daten, die (nach ihren Eigenschaften, Werten, Kriterien) in Listen vorsortiert und mit Verweisen verkettet sind, *sequentiell*.

Alternativ dazu haben wir einen Datenbestand in Form eines *Feldes*. Das setzt voraus, daß es ein Kriterium (Eigenschaft, Wert) der Daten gibt, das sich zu einer eindeutigen Speicherabbildung eignet. Der gesamte Speicherbereich, der dem Wertebereich der Adreßfunktion entspricht, bildet das Feld; er ist für diejenigen Adressen, denen aktuell kein vorgegebenes Datum entspricht, mit einer Codierung für "nicht vorhanden" besetzt. In unserem konkreten Fall werden zum Adressieren Knotennummern irgendeiner vorgegebenen Ordnung benutzt, und wir haben Vektoren (für die Datenbestände der Knoten) und Matrizen (für die Datenbestände der Kanten, betrachtet als geordnete Knotenpaare). Wir bearbeiten diese Vektoren und Matrizen mit *Paralleloperationen* im Sinne der Vektoren-Arithmetik in Kapitel 3, das heißt wir verarbeiten adressierbare Vektoren bekannter Länge.

Sollen Feld-Algorithmen auf einem ausschließlich assoziativen System bearbeitet werden, muß die "äußere" Adressierung umgesetzt - zurückübersetzt - werden in eine *inhaltliche Adressierung* wie zum Beispiel beim Hamiltonalgorithmus, wo wir $n!$ Zellen vorbesetzen mit allen möglichen Permutationen der Zahlen $1, 2, \dots, n$ in geeigneter Codierung. Das kostet aber unter anderem einen hohen Aufwand für die Vorbereitung, etwa das Einlesen eines solchen "Adressierschemas" vom Hintergrundspeicher.

Bei Datei-Algorithmen muß - zumindest bei jedem auswählenden Zugriff - prinzipiell der gesamte Datenbestand bearbeitet werden. Ob bei der eigentlichen Verarbeitung ein assoziatives System ein gewisses Maß an Parallelisierung bringen kann, hängt

vom Algorithmus ab. Bei den Algorithmen für die Eliminationsgraphen, die vom Typ "depth-first-search" sind, und dem Hamilton-Algorithmus nach dem Back-track-Verfahren kann nicht parallelisiert werden, weil *assoziative Systeme nicht nach mehreren Suchschlüsseln zugleich suchen können*.* Hier haben also sequentielle und assoziatives System dasselbe Laufzeitverhalten, während für das mikroprogrammierte Parallel- und Assoziativkonzept für die Laufzeit ein Faktor von der Größenordnung des Datenumfanges dazukommt; bei den Abschätzungen dort bin ich aber stillschweigend davon ausgegangen, daß etwa bei sequentieller Verarbeitung einzelner Daten nicht mit einer 1-Bit-Maske über die Breite des gesamten Datenbestandes "parallel" gearbeitet wird, sondern vielleicht sogar mit dem konventionellen Befehlssatz des ursprünglichen (sequentiellen) Maschinenkonzeptes.

Die Feld-Algorithmen wurden angeregt durch die Algorithmen für zellulare Arrays. Diese Algorithmen konnten nachgebildet werden mit qualitativ vergleichbaren Laufzeiten (mit Ausnahme des "schnellen" Algorithmus zur Bestimmung eines aufspannenden Baumes, dessen zellulares Array von erheblicher Komplexität ist und beim Nachbilden viele Sequentialisierungen erfordert) und beliebig vielen beliebig formatierten Daten. Dabei zeigt sich, daß die mikroprogrammierte Version überall dort qualitativ (abstrakt) schneller ist, wo die Anzahl von Operationen etwa von der Anzahl der Einsen in der Adjazenzmatrix abhängig ist - während die Laufzeit in der Hardware des zellularen Arrays unabhängig davon ist, ob 0 oder 1 verarbeitet wird. Bei den Algorithmen für Eliminationsgraphen ergab sich teils eine Verbesserung, weil die Darstellung dem ursprünglichen Problem besser angepaßt ist, teils aber eine leichte Verzögerung (bei der Bestimmung der minimalen Ordnung), weil hier nun der sequentielle Algorithmus nur genau soviel Schritte macht, wie Datenkombinationen vorhanden sind, während der Matrixalgorithmus schematisch arbeitet. Der Datenumfang einer Paralleloperation des mikroprogrammierten Konzeptes bei Verarbeitung von Matrizen ist im allgemeinen eine Matrixzeile, gelegentlich aber auch, etwa bei der Maximumsuche für den maximalen aufspannenden Baum, die gesamte Matrix. Feld-Algorithmen sind umso effektiver, je dichter der Graph ist, d.h. je mehr Kanten er enthält.

Im Gegensatz zum Assoziativkonzept erlaubt das Parallelkonzept das *mehrfache Kopieren umfangreicher Datenblöcke* (Vektoren), wie es beim Hamiltonalgorithmus gezeigt wurde. Diese Möglichkeit läßt sich aber dort nicht effektiv ausnutzen,

*) Das gilt, wie schon mehrfach erwähnt, unabhängig von der Realisierung der assoziativen Operationen.

wo es sich um ständig "dünner" werdende Datenbestände handelt wie in diesem Beispiel. Hier würde nur helfen, das rigide Konzept der Parallelverarbeitung aufzubrechen und zu erweitern durch Operationen mit Packen, was einem Übertragen mit "Garbage Collection" entspricht.

Die Entscheidung zwischen den beiden möglichen Implementierungen des mikroprogrammierten Parallel- und Assoziativkonzeptes läßt sich nicht als Ergebnis der Vergleiche, wie sie in Tabelle 4.1 zusammengefaßt sind, gewinnen. Es ist aber festzustellen, daß speziell die Matrixalgorithmen erfolgreich bearbeitet werden können mit Hilfe der Zusatz-Hardware zur Unterstützung der Vertikalverarbeitung: dem orthogonalen Datenzugriff zur Transposition von Bit-Matrizen und zum Lesen einer Bitmatrix-Spalte zum Weiterverwenden als "Relevanzmaske"; dem Skalierbefehl zum effektiven Auswerten von "Masken" als Bedingungen für sequentielle Mehrfachoperationen. Insgesamt erscheint auch die Vertikalverarbeitung besser geeignet als die konventionelle Emulation, Masken konsistent zu verarbeiten. Soll das Parallelkonzept erweitert werden, wie oben skizziert, um "Übertragen mit Garbage Collection", so steht hierfür entweder mit dem Orthogonalspeicher oder, in einem System mit Drehscheibe, mit dem Quersummenmodul geeignete Hardware zur Verfügung.

4.3.2 Bedeutung des mikroprogrammierten Parallel- und Assoziativkonzeptes für die Graphenalgorithmen

1. In einem System mit konventioneller Basis-Hardware - eventuell ergänzt um einige Zusatz-Komponenten - haben wir unkonventionelle Operationen zur Verfügung zur Programmierung nicht-numerischer nicht-sequentieller Algorithmen, wobei vorausgesetzt sei, daß diese Operationen mit Hilfe einer geeigneten höheren Programmiersprache handhabbar seien. Die Algorithmen sind im allgemeinen kompakter programmierbar als sequentielle Algorithmen, erfordern etwa kein Vorsortieren von Datenmengen. Ihre Laufzeit ist allerdings, in realer Maschinenzeit gemessen, länger.

2. Speziell das in Kapitel 3 ausführlich entwickelte Konzept erlaubt durch seine Kombination assoziativer und paralleler Operationen, ein größeres Spektrum unkonventioneller Algorithmen zu implementieren, als etwa ein ausschließlich assoziatives System. Zusätzlich steht weiterhin, wo es gebraucht wird, das konventionelle System der Grundausstattung zur Verfügung.

3. Jedes solche mikroprogrammierte System stellt ein Experimentiersystem dar: Einerseits kann man ausprobieren, ob ein System überhaupt brauchbar ist zur Lösung einer oder mehrerer vorgegebener Aufgabenklassen, andererseits kann man Algorithmen und ganz allgemein Erfahrungen sammeln schon zu einer Zeit, wo die Hardware-Realisierung eines Systems noch auf sich warten läßt. Das *Parallelkonzept* hat sich insgesamt als brauchbar erwiesen für alle skizzierten Algorithmen. Beim - hardware- oder firmware-realisierten - *Assoziativkonzept* hat sich aber gezeigt, daß wesentliche notwendige Operationen nicht realisierbar sind: Suchen nach mehreren Schlüsseln gleichzeitig und Kopieren von Datenblöcken. Die wesentliche Vereinfachung gegenüber sequentiellen Algorithmen war nur - will man nicht Techniken wie etwa gespeicherte Adreßschemata anwenden, wie beim Hamilton-Algorithmus skizziert -, daß Daten nicht vorsortiert und in Listen verwaltet zu werden brauchen. Zumindest für die hier betrachteten Algorithmen lohnt es also *nicht*, große Erwartungen an Systeme mit "großen" assoziativen Speichern zu stellen.

Anhang A: Konventionelle Emulation des Parallel- und Assoziativkonzeptes

Hier sollen einige horizontale Blockoperationen als Implementierungsskizzen zusammengestellt werden und zwar solche, die beim Vergleich besonders deutlich die typischen Unterscheidungen zwischen Horizontal- und Vertikalverarbeitung hervortreten lassen.

Allgemein gilt: Die Auswahl- und die Überlaufmarkierung müssen bei horizontaler Darstellung in (zwei oder) einer Liste stehen, auf die man zum Beispiel durch Indexwerte zugreifen kann. Eventuell können diese Informationen in mehreren Speicherworten kompakt zusammengefaßt werden um den Preis eines komplizierteren Zugriffs.

A.1 Addition

sequ $i = 0, 1, \dots, n-1$

"Schleife über alle Operandenpaare"

if m_i equal 1

"Abfrage, ob ausgewählt"

then sequ $k = q-1, q-2, \dots, 0$

"Laufschleife über alle Teilworte einer Zahlendarstellung"

par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

read $a_{i,kr+j};$

read $b_{i,kr+j};$

"lesen je ein Teilwort der beiden Operanden"

sim $\bar{s}_i^{(k)} := \text{addwithcarry } (\bar{a}_i^{(k)}, \bar{b}_i^{(k)})$

with $c := \dots$

with $o := \dots$

end sim;

"konventionelle Addition unter Einschluß des Übertrags mit Erzeugung von Flags; implizite Voraussetzung: durch Indexrechnung für Schleife ist der Übertrag bei $k=q-1$ gelöscht"

```

        write  $s_{i,kr+j}$ 
    end par;
     $o_i := 0$ 
end sequ
write  $o_i$ 
"schreiben Überlaufmarkierung"

else
end if

end sequ

```

A.2 Multiplikation

sequ $i = 0, 1, \dots, n-1$

"Laufschleife über alle Operandenpaare"

if $m_i \text{ equal } 1$
 "Abfrage, ob ausgewählt"

then $\overline{s(0)} := \text{null};$
 $\overline{s(1)} := \text{null};$
 $\overline{s(2)} := \text{null};$

"vorbesetzen drei Hilfsregister für Produktzwischenwerte"

sequ $k = 2q-2, 2q-3, \dots, 1$

"Laufschleife über alle Kombinationen von Teilworten der Zahlendarstellung der beiden Operanden mit Ausnahme der Kombination der beiden höchstwertigen Teilworte"

sequ $h = q-1, q-2, \dots, 0$

"Laufschleife über alle Teilworte der Zahlendarstellung des ersten Operanden"

if $h \text{ notgreater } k$

then par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerks"

read $a_{i,hr+j};$

read $b_{i,(k-h),r+j};$

"lesen je ein Teilwort der beiden Operanden"

sim $\overline{x_i(\emptyset)} := \text{multiplyunsigned}(\emptyset)(\overline{a_i^{(h)}}, \overline{b_i^{(k-h)}})$
with $\overline{x_i(1)} := \text{multiplyunsigned}(1)(\overline{a_i^{(h)}}, \overline{b_i^{(k-h)}})$

end sim;

"berechnen obere und untere Hälfte des Teilprodukts ohne Berücksichtigung von Vorzeichen; konventionelle Festkomma-Arithmetik"

sim $\overline{s(2)} := \text{add}(\overline{s(2)}, \overline{x_i(1)})$

with $c := \dots$

end sim;

"aufaddieren untere Hälfte des Teilproduktes"

sim $\overline{s(1)} := \text{addwithcarry}(\overline{s(1)}, \overline{x_i(\emptyset)})$

with $c := \dots$

end sim;

"aufaddieren obere Hälfte des Teilproduktes"

sim $\overline{s(\emptyset)} := \text{addwithcarry}(\overline{s(\emptyset)}, \text{null})$

with $c :=$

end sim

"aufaddieren Übertrag"

end par

else

end if

end sequ;

write $\overline{s(2)}$;

"schreiben unterstes Teilwort des Teilergebnisses"

$\overline{s(2)} := \overline{s(1)}$;

$\overline{s(1)} := \overline{s(\emptyset)}$;

$\overline{s(\emptyset)} := \underline{\text{null}}$

"verschieben bisheriges Teilergebnis um ein Wort"

end sequ;

par $j = \emptyset, 1, \dots, n-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

read $a_{i,j}$;

read $b_{i,j}$;

"lesen höchstwertige Teilworte der beiden Operanden"

sim $\overline{x}(\emptyset) := \text{multiply}(\emptyset) (\overline{a_i}(\emptyset), \overline{b_i}(\emptyset))$

with $\overline{x}(T) := \text{multiply}(1) (\overline{a_i}(T), \overline{b_i}(T))$

end sim;

"berechnen obere und untere Hälfte des Teilproduktes mit Berücksichtigung des Vorzeichens, konventionelle Festkomma-Arithmetik"

sim $\overline{s}(2) := \text{add}(\overline{s}(2), \overline{x}(T))$

with $c := \dots$

end sim;

"aufaddieren untere Hälfte des Teilproduktes"

write $\overline{s}(2)$;

sim $\overline{s}(T) := \text{addwithcarry}(\overline{s}(T), \overline{x}(\emptyset))$

with $c := \dots$

end sim;

"aufaddieren obere Hälfte des Teilproduktes"

write $\overline{s}(T)$

end par

else

end if

end sequ

A.3 Maximumsuche

sequ $k = \emptyset, 1, \dots, q-1$

"Laufschleife über alle Teilworte einer Zahlendarstellung"

par $j = \emptyset, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

set $j := \emptyset$

"vorbesetzen des Maximums mit der kleinstmöglichen Zahl, hier noch ohne ihr Vorzeichen"

end par

end sequ;

$x_0 := 1;$

"vorbesetzen negatives Vorzeichen"

sequ $i = 0, 1, \dots, n-1$

"Laufschleife über alle Operanden"

if m_i equal 1

"Abfrage, ob ausgewählt"

then sequ $k = 0, 1, \dots, q-1$

"Laufschleife über alle Teilworte einer Zahlendarstellung"

par $j = 0, 1, \dots, n-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

read $a_{i,kr+j};$

sim $\bar{y} := \text{subtract } (\bar{s}^{(k)}, \bar{a}_i^{(k)})$

with $r : \dots$

end sim

"konventionell subtrahieren"

end par

end sequ;

if y_0 equal 1

"Abfrage, ob Operand größer als bisheriges Maximum"

then sequ $k = q-1, q-2, \dots, 0$

"Laufschleife über alle Teilworte einer Zahlendarstellung;

par $j = 0, 1, \dots, n-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

read $a_{i,kr+j};$

$s_{kr+j} := a_{i,kr+j};$

"übernehmen größeren Wert als Maximum"

write $\bar{x};$

"notieren Index des neuen Maximums"

end par

else

end if

else

end if

end sequ;

sequ $k = 0, 1, \dots, q-1$

"Laufschleife über alle Teilworte einer Zahlendarstellung"

par $j = 0, 1, \dots, n-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

$\text{write } s_{kr+j}$

"schreiben Maximum-Wert"

end par

end sequ

A.4 "Best Match"

par $j = 0, 1, \dots, n-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

$s_j := 0$

"vorbesetzen Maximum mit der kleinstmöglichen Zahl, ohne Vorzeichen"

end par;

$\delta_0 := 1;$

"vorbesetzen negatives Vorzeichen"

sequ $i = 0, 1, \dots, n-1$

"Laufschleife über alle Operanden"

if $m_i \text{ equal } 1$

"Abfrage, ob ausgewählt"

then par $j = 0, 1, \dots, n-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

$y_j := 0$

end par;

sequ $k = 0, 1, \dots, q-1$

"Laufschleife über alle Teilworte einer Zahlendarstellung"

par $j = 0, 1, \dots, n-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

$\text{read } a_{i,kr+j};$

$x_{kr+j} := a_{i,kr+j} \odot b_{i,kr+j}$

$x_{kr+j} := \neg x_{kr+j}$

$x_{kr+j} := x_{kr+j} \wedge e_{kr+j}$

"erzeugen Markierung, wo Übereinstimmung"

$\bar{y} := \text{sum}(x_{kr+j}, e_{kr+j})$ *

"aufsummieren Bitwerte"

end par

end sequ;

par $j = 0, 1, \dots, r-1$

"normale Parallelarbeit des Speichers bzw. Rechenwerkes"

$\bar{z} := \text{subtract}(\bar{y}, \bar{s})$;

"konventionelle Festkommasubtraktion zum Vergleich mit bisherigem Maximum"

if $\bar{z}$ greater null

"Abfrage, ob neuer Wert größer"

then $\bar{s} := \bar{y}$;

write $\bar{z}$

"notieren Index des neuen Maximums"

else

end if

end par

else

end if

end sequ

*) genaue Beschreibung in 3.5.2.2

Anhang B: Numerische Anwendungsbeispiele für das Parallel- und Assoziativkonzept: Potenzberechnung

Kung /29/ gibt folgenden Algorithmus an zur Berechnung der n-ten Potenz einer vorgegebenen Zahl x_0 :

Sei $a \neq 0$ eine beliebige Zahl; a_i , $i = 1, 2, \dots, n$ seien die Wurzeln von $x^n - a$, $a_i \neq a_j$ für $i \neq j$ und $a_i \neq x_0$ für alle i ; sei

$$b_i = \frac{a_i}{n \cdot a}, \quad i = 1, 2, \dots, n$$

Der Algorithmus (ohne Deklarationen):^{*}

par $i = 1, 2, \dots, n$

$c_i := \text{subtract } (x_0, a_i);$

$d_i := \text{divide } (b_i, c_i);$

$e := \text{sum } (d_i)$

end par;

$f := \text{divide } (1, e);$

$g := \text{add } (f, a)$

"g ist das Ergebnis, also x_0^n "

Die algebraischen Grundlagen des Algorithmus lassen sich kurz zusammenfassen:

$$P(x) = x^n - a = \prod_{i=1}^n (x - a_i)$$

$$P'(a_j) = n \cdot a_j^{n-1} = \prod_{i \neq j} (a_j - a_i)$$

$$\text{Sei } Q(x) = \sum_{j=1}^n a_j \prod_{i \neq j} (x - a_i)$$

$$Q(a_j) = a_j \prod_{i \neq j} (a_j - a_i) = a_j \cdot P'(a_j) = n \cdot a_j^n \equiv n \cdot a \equiv Q(x)$$

^{*}) Notation ähnlich der in Kapitel 3, aber die Variablen hier sind Festkommazahlen.

Dann ist also:

$$e = \sum_{i=1}^n d_i = \sum_{i=1}^n \frac{a_i}{n \cdot a \cdot (x_0 - a_i)} = \frac{Q(x_0)}{na \cdot P(x_0)} = \frac{1}{P(x_0)}$$

$$f = \frac{1}{e} = P(x_0)$$

$$g = f + a = P(x_0) + a = x_0^n - a + a = x_0^n$$

Kung/29/ gibt als Abschätzung für die Rechenzeit für ein System mit k parallel arbeitenden Prozessoren für $k \geq n$ an:

$$T_k(x^n) \leq \lceil \log_2 n \rceil \cdot A + 2(A+D) \quad (1)$$

Also im allgemeinen:

$$T_k(x^n) \leq \lceil \log_2 n \rceil \cdot M$$

Mit A Operationszeit für eine Addition, D Operationszeit für eine Division, M eine Operationszeit für eine Multiplikation und der zusätzlichen Voraussetzung

$$A < M$$

Für die Summation werden dabei $\lceil \log_2 n \rceil$ paarweise Additionen benutzt.

Für unser mikroprogrammiertes Parallel- und Assoziativkonzept haben wir als Abschätzung, wenn $k \leq r$ (Rechenwerkbreite):

$$T(x^n) \leq A_v + D_v + S_v + D_s + A_s$$

A, D wie oben, S Operationszeit für die Summation, der Index v bedeutet "Vektor", der Index s bedeutet "Skalar". Mit

$$S_v \leq A_v$$

und unter Vernachlässigung der Unterscheidung nach Vektor- und Skalarverknüpfung haben wir die Abschätzung

$$T(x^n) \leq A + 2 \cdot (A+D) \quad (2)$$

Qualitativ bedeutet das also für $n > 2$ einen Zeitgewinn durch die Konstruktion der Summenoperation.

Literatur

- / 1/ V.R. Basili: "Some Supplementary Notes on the Graph Algorithmic Language GRAAL", in: "Graphensprachen und Algorithmen auf Graphen", 1. Fachtagung über Graphentheoretische Konzepte der Informatik, Berlin, 1975, herausgegeben von U. Pape, Carl Hanser Verlag, München, Wien, 1976
- / 2/ D. Basu, T. Jayashree: "On a Simple Postcorrection for Nonrestoring Division", IEEE Transactions on Computers, C-24 (1975), 1019-1020
- / 3/ J.R. Bunch: "Complexity of Sparse Elimination", Symposium on Complexity of Sequential and Parallel Numerical Algorithms, held at Carnegie-Mellon University, edited by J.F. Traub, Academic Press, New York and London, 1973
- / 4/ N. Christofides: "Graph Theory. An Algorithmic Approach", Academic Press, New York, London, San Francisco, 1975
- / 5/ J.W. Cooley, J.W. Tuckey: "An Algorithm for the Machine Calculation of Complex Fourier Series", Mathematics of Computation, 19 (1965), 297-301
- / 6/ B.A. Crane: "Path Finding with Associative Memory", IEEE Transactions on Computers, C-19 (1968), 691-693
- / 7/ S.H. Dalrymple: "A Parametric Associative Memory Emulation", Workshop on Computer Architecture for Non-numeric Processing, held at Southern Methodist University, Dallas, Texas, 1974, MDAC Paper WD 2441, May 1975
- / 8/ R. Eck: "Erstellung und Simulation von Vertikalbefehlen für den Mini-rechner SUE", Diplomarbeit am IMMD, Erlangen 1976
- / 9/ A. Endres: "Formale Analyse und Verifikation von Programmen", Stuttgart, 1976, Dissertation
- /10/ T. Feng: "Large Scale Information Processing Systems, Volume V. Study of Associative Memory Systems", Syracuse University, Final Technical Report, RADR-TR-70-80, Volume V, May 1970

- /11/ P.M. Flanders, D.J. Hunt, D. Parkinson, S.F. Reddaway: "Experience Gained in Programming the Pilot DAP, a Parallel Processor", IMACS(AICA)-GI-Symposium on Parallel Computers - Parallel Mathematics, München, 1977, to appear
- /12/ I. Flores: "The Logic of Computer Arithmetic", Prentice-Hall, Englewood Cliffs, 1963
- /13/ H.H. Guild: "Some Cellular Logic Array for Non-Restoring Binary Division", The Radio and Electronic Engineer, 39 (1970), 345-348
- /14/ W. Händler: "Rechenwerk einer digitalen Rechenmaschine", Patentschrift 1157009, Telefunken Patentverwertungsgesellschaft mbH, Ulm/Donau, 13.9.1961
- /15/ W. Händler: "Unconventional Computation by Conventional Equipment", Arbeitsbericht IMMD, Band 7, Nummer 2, Erlangen, 1974
- /16/ W. Händler, F. Hofmann, H.J. Schneider: "A General Purpose Array with a Broad Spectrum of Applications", Workshop on Computer Architecture, held at Erlangen 1975, Springer-Verlag, Berlin, Heidelberg, New York, 1976
- /17/ W. Händler, R. Klar: "Fitting Processors to the Needs of a General Purpose Array (EGPA)", Workshop MICRO-8, Chicago, 1975
- /18/ W. Händler: "Prozessor mit Mikroprogrammsteuerung einer digitalen Rechenanlage", Patentschrift 2419241, 24.2.1977
- /19/ H. Hessenauer: "Support of Vertical Data Processing by Additional Hardware", IMACS (AICA)-GI-Symposium on Parallel Computers - Parallel Mathematics, München, 1977, to appear
- /20/ L.C. Higbie: "Vector Floating-Point Data Format", IEEE Transactions on Computers, C-25 (1976), 25-32
- /21/ W. Hoffmann, H. Wichmann: "Konzept der Vertikalverarbeitung. Implementierung und Analyse vertikaler Algorithmen auf einem mikroprogrammierbaren Rechner", Arbeitsbericht IMMD, Band 9, Nummer 3, Erlangen, 1976

- /22/ W. Hoffmann: "Implementation and Evaluation of Vertical Algorithms on a Microprogrammable Computer", IMACS (AICA)-GI-Symposium on Parallel Computers - Parallel Mathematics, München, 1977, to appear
- /23/ T.C. Hu: "Revised Matrix Algorithms for Shortest Paths", SIAM Journal on Applied Mathematics, 15 (1967), 207-218
- /24/ T. Ito: "An Algebraic Theory of Pattern Manipulation", Second International Joint Conference on Pattern Recognition, Copenhagen, 1974
- /25/ B. Jahn: "Eine Studie zur Implementierung von Graph-Grammatiken unter Verwendung von Assoziativverarbeitung", Diplomarbeit am IMMD, Erlangen, 1976
- /26/ D.B. Johnson: "Efficient Algorithms for Shortest Paths in Sparse Networks", Journal of the ACM, 24 (1977), 1-13
- /27/ A.K. Kamai, H. Singh, D.P. Agrawal: "A Generalized Pipeline Array", IEEE Transactions on Computers, C-23 (1974), 533-536
- /28/ B. Kruse: "A Parallel Picture Processing Machine", IEEE Transactions on Computers, C-22 (1973), 1075-1087
- /29/ H.T. Kung: "New Algorithms and Lower Bounds for the Parallel Evaluation of Certain Rational Expressions and Recurrences", Journal of the ACM, 23 (1976), 252-261
- /30/ L. Lamport: "The Parallel Execution of DO Loops", Communications of the ACM, 17 (1974), 83-93
- /31/ K.N. Levitt, W.H. Kautz: "Cellular Arrays for the Solution of Graph Problems", Communications of the ACM, 15 (1972), 789-801
- /32/ J.C. Majithia, R. Kitai: "An Iterative Array for Multiplication of Signed Binary Numbers", IEEE Transactions on Computers, C-20 (1971), 214-216
- /33/ J.C. Majithia, R. Kitai: "A Cellular Array for the Nonrestoring Extraction of Square Roots", IEEE Transactions on Computers, C-20 (1971), 1617-1618

- /34/ J.C. Majithia: "Cellular Array for Extraction of Squares and Square Roots of Binary Numbers", IEEE Transaction on Computers, C-21 (1972), 1023-1024
- /35/ V. Mierlea: "An Algorithm for Scheduling n Works on m Machines", Economic Computation and Economic Cybernetics Studies and Research, 4 (1971), 17-28
- /36/ M. Minsky, S. Papert: "On Associative, Parallel, and Analog Computations", Symposium on Associative Information Techniques, held at General Motors Research Laboratories, Warren, Michigan, 1968, edited by E.L. Jacks, Elsevier, New York, 1971
- /37/ H. Noltemeier: "Graphentheorie mit Algorithmen und Anwendungen", W. de Gruyter, Berlin, New York, 1976
- /38/ U. Pape: "Zur Implementierung und Wirtschaftlichkeit von MOORE-Algorithmen zur Bestimmung kürzester Weglängen in einem Netzwerk", Bericht Nr. 73-o6, Technische Universität Berlin, Fachbereich 20 - Kybernetik, 1973
- /39/ U. Pape: "Graphen-Sprachen - eine Übersicht", in: "Graphen-Sprachen und Algorithmen auf Graphen", 1. Fachtagung über Graphentheoretische Konzepte der Informatik, Berlin, 1975, herausgegeben von U. Pape, Carl Hanser Verlag, München, Wien, 1976
- /40/ U. Pape: "Datenstrukturen für Mengen in Algorithmen auf Graphen", in: "Graphen, Algorithmen, Datenstrukturen", 2. Fachtagung über Graphentheoretische Konzepte der Informatik, Göttingen, 1976, herausgegeben von H. Noltemeier, Carl Hanser Verlag, München, Wien, 1976
- /41/ U. Pape: "A Model of a High Level Design Language for Graph Algorithms", in: "Graphen, Algorithmen, Datenstrukturen", 2. Fachtagung über Graphentheoretische Konzepte der Informatik, Göttingen, 1976, herausgegeben von H. Noltemeier, Carl Hanser Verlag, München, Wien, 1976
- /42/ U. Pape: "Eine Bibliographie zu "Kürzeste Weglängen und Wege in Graphen und Netzwerken"", Bericht 77-o7 der Lehrereinheit für Angewandte Elektronische Datenverarbeitung der Technischen Universität Berlin, 1977

- /43/ B. Parhami: "Associative Memories and Processors: An Overview and Selected Bibliography", Proceedings of the IEEE, 61 (1973), 722-730
- /44/ U.W. Pooch, A. Nieder: "A Survey of Indexing Techniques for Sparse Matrices", Computing Surveys, 5 (1973), 109-133
- /45/ S.F. Reddaway: "DAP - A Distributed Array Processor", First Annual Symposium on Computers Architecture, 1973, edited by G.J. Lipovski, S.A. Szygenda
- /46/ W. Reimer: "Vergleichende Simulation von Befehlen für vertikale Informationsverarbeitung an verschiedenen Prozessoren", Diplomarbeit am IMMD, Erlangen, 1975
- ,47 R.K. Richards: "Arithmetic Operations in Digital Computers", D. van Norstrand Company, Princeton, New York, 1955
- /48/ D.J. Rose, R.E. Tarjan, G.S. Lueker: "Algorithmic Aspects of Vertex Elimination on Graphs", SIAM Journal of Computings, 5 (1976), 266-283
- /49/ A. Rosenfeld: "Connectivity in Digital Pictures", Journal of the ACM, 17 (1970), 146-160
- /50/ D.A. Savitt, H.H. Love, R.E. Thoop: "ASP: A New Concept in Language and Machine Organization", AFIPS 1967 Spring Joint Computer Conference, 30 (1967)
- /51/ W. Shooman: "Orthogonal Processing", in: L.C. Hobbs (editor): "Parallel Processor Systems, Technologies, and Applications", Spartan Books, New York, 1970
- /52/ STARAN, Reference Manual, GER-1563A, Goodyear Aerospace Corporation, September 1973
- :3 H. Steckert: "Einbettung eines zusätzlichen Moduls zur Unterstützung der Vertikalverarbeitung in den Rechner INTERDATA 85", Diplomarbeit am IMMD, Erlangen, 1976

- /54/ R. Tarjan: "Depth-First Search and Linear Graph Algorithms", SIAM Journal of Computing, 1 (1972), 146-160

- /55/ K.J. Thurber, L.D. Wald: "Associative and Parallel Processors", Computing Surveys, 7 (1975), 215-255

- /56/ M.A. Wesley, S.-K. Chang, J.H. Mommens: "A Design for an Auxiliary Associative Parallel Processor", AFIPS 1972 Fall Joint Computer Conference, 41 (1972), 461-472

- /57/ H. Wichmann: "Algorithms for Vertical Processing", IMACS (AICA)-GI-Symposium on Parallel Computers - Parallel Mathematics, München, 1977, to appear

Liste der Identifikatoren in den Mikroprogramm-Algorithmen

Allgemeine Bemerkung: Die Indizes der Bits in den Vektoren beziehen sich auf die unsegmentierte Bitmatrix (vgl. Seiten 31 und 32), nicht auf ihre Interpretation.

$a_{i,j}$	(erster) Operandenvektor, horizontal: Komponente i, Bit j; vertikal: Komponente j, Bit i
$\bar{a}_i^{(k)}$	(erster) Operandenvektor, nur horizontal: k-tes Teilwort der i-ten Komponente bei konventioneller Festkomma-darstellung
$b_{i,j}$	zweiter Operandenvektor, siehe $a_{i,j}$
$\bar{b}_i^{(k)}$	zweiter Operandenvektor, siehe $\bar{a}_i^{(k)}$
c	Übertrag-Markierung (Carry-Flag) bei konventionellen Festkomma-operationen
$c_{i,j}$	Übertrag, nur vertikal: Komponente j, Übertrag aus Bit i
d_j	Operationencodierung zur Auswahl innerhalb der Funktionenpaare Addition/Subtraktion bzw. Zweier-/Einerkomplement bzw. Maximum/Minimum, nur vertikal: Komponente j
$\bar{d}$	Operationencodierung, nur Vertikal: Zusammenfassung der Bits für alle Komponenten
e_i	Relevanzmaske, horizontal und vertikal: Bit i
f_j	Operationencodierung zur Auswahl innerhalb des Funktionenpaares lexikalische/arithmetische Suche, siehe d_j
$\bar{f}$	Operationencodierung, siehe $\bar{d}$
h	Index
i	Index auf Zeile in Bitmatrix
$\bar{i}$	Indexwert als Festkommazahl
j	Index auf Spalte in Bitmatrix
k	Index
ℓ	Anzahl der Bits zur Darstellung einer Komponente
m_j	Auswahlmaske, horizontal und vertikal: Komponente j
$\bar{m}$	Auswahlmaske, nur vertikal: Zusammenfassung der Bits für alle Komponenten
$\bar{m}^{(k)}$	Auswahlmaske, nur vertikal: Zusammenfassung der Bits für die Komponenten des k-ten Teilvektors (Teilblockes) à r Komponenten

$m(=)_j$	Ergebnismaske bei komponentenweisem Vergleich mit Ergebnis "gleich", nur vertikal: Komponente j
$\overline{m(=)}$	Ergebnismaske, nur vertikal: Zusammenfassung der Bits für alle Komponenten
$m(>)_j$	Ergebnismaske mit Ergebnis "größer", siehe $m(=)_j$
$\overline{m(>)}$	Ergebnismaske, siehe $\overline{m(=)}$
$m(<)_j$	Ergebnismaske mit Ergebnis "kleiner", siehe $m(=)_j$
$\overline{m(<)}$	Ergebnismaske, siehe $\overline{m(=)}$
n	Anzahl der Komponenten in einem Vektor (Datenblock)
o	Überlauf-Markierung (Overflow-Flag) bei konventionellen Festkommaoperationen
o_j	Überlauf-Markierung, horizontal und vertikal: Komponente j
$p = \left\lceil \frac{n-1}{r} \right\rceil$	Anzahl der Teilvektoren (Teilblöcke) à r Komponenten bei Vertikaldarstellung eines Vektors
$q = \left\lceil \frac{\ell-1}{r} \right\rceil$	Anzahl der Teilworte à r Bits bei Horizontaldarstellung einer Komponente
r	Speicherwortlänge und Rechenwerkbreite
$s_{i,j}$	Ergebnisvektor, siehe $a_{i,j}$
$\bar{s}_i(k)$	Ergebnisvektor, siehe $\bar{a}_i(k)$
$\bar{s}$	Ergebniswert bei Komponentensumme in konventioneller Festkomma-darstellung
$\bar{s}^{(k)}$	Ergebniswert, k-tes Teilwort bei konventioneller Festkommadarstellung
$\overline{s(\emptyset)}$	Zwischenwert bei Produktberechnung, nur horizontal: einfache Festkommazahl
$\overline{s(1)}$	Zwischenwert bei Produktberechnung, siehe $\overline{s(\emptyset)}$
$\overline{s(2)}$	Zwischenwert bei Produktberechnung, siehe $\overline{s(\emptyset)}$
u	Hilfsvektor für "Best Match", nur vertikal
$u_{i,j}$	Hilfsvektor für "Best Match", nur vertikal: Komponente j, Bit i
$\bar{x}, x_j, x_{i,j}, \overline{x(\emptyset)}, \overline{x_i(\emptyset)}, \overline{x(1)}, \overline{x_i(1)}, y_j, \bar{y}, z_j, \bar{z}$ sind Zwischenergebnisse.	

Register

Addierer-/Subtrahierer-Zelle eines Pipeline-Arrays	26
Addition (konventionell)	151
Addition/Subtraktion	41, 92, 93, 96
Addition/Subtraktion (Vertikalverarbeitung)	40, 41
Aktivitätsmaske	38
algebraische Graphenalgorithmen	102, 140
arithmetische Pipeline-Arrays	17, 24, 71, 72
arithmetisches Maximum/Minimum (Vertikalverarbeitung)	58
assoziative Operationen	4
assoziativer Prozessor	20
assoziativer Speicher	18, 142, 150
assoziative Systeme	18, 71, 118, 129, 135, 142, 144, 150
assoziative Verarbeitung	63, 129, 135
Auffüllen eines Eliminationsgraphen	124
aufspannender Baum eines Graphen	104, 106, 115
Auswahlmaske	38, 78
Back-track-Verfahren	139, 148
Befehlsliste der assoziativen Verarbeitung	64
Befehlsliste der Vektoren-Arithmetik	39
beschreibbarer Speicherspeicher	10
Beschreibungssprache	33
"Best Match"	19, 77
"Best Match" (konventionell)	156
"Best Match" (Vertikalverarbeitung)	66
Booth'scher Multiplikationsalgorithmus	41, 45
DAP	7, 17
Datei	108, 147
Datei-Algorithmen	107, 147
Datenkonversion	6, 13
Datenstrukturen für Graphen-Algorithmen	109
Definition (Deklaration) von Datenstrukturen	108
Depth-first-search	124, 148
Distanzmatrix	121
Division	51
Division (Pipeline-Array)	25
Division (Vertikalverarbeitung)	51
"Drehscheibe"	12, 14, 69, 111, 119
dünne Matrizen	101, 123
Echtzeit-Verarbeitung	6
Einerkomplement (Vertikalverarbeitung)	44
"einfacher" Multiplikationsalgorithmus	48
Einschreiben (Vertikalverarbeitung)	68
Einzeldaten-Operationen	29
Ein-/Ausgabe-Register des assoziativen Speichers	18
Eliminationsgraph	102, 123
Eliminationsordnung	124
enumerative Graphenalgorithmen	102
Erweiterbarkeit der Hardware	11
"Exact Match"	18
"Exact Match" (Vertikalverarbeitung)	64
Extremumsuche	58, 77

Fast Fourier Transform	37, 99
Feld	109
Feld-Algorithmen	107, 147
Fundamentalzyklen zu aufspannendem Baum	104, 115
funktionaler Speicher	21
Gesamtdaten-Operationen	29
Gleitkomma-Arithmetik	38
Graphen-Sprachen	107
Hamilton-Wege	102, 138
Hilfsoperationen	70
höhere Programmiersprache	13
homogene Algorithmen	2, 17, 29
horizontal codierte Mikrooperationen	9
Hu's Algorithmus	120
Klassifikation der Operationen	29
Kommentar	11
Komponentensumme	54, 77
Komponentensumme (Vertikalverarbeitung)	54
Komponentenweiser Vergleich (Vertikalverarbeitung)	56
Konversion	6, 13
kürzeste Wege in einem Graphen	117
lexikalisches Maximum/Minimum (Vertikalverarbeitung)	59, 69
lineare Operationen	1, 29
Maskenregister des assoziativen Speichers	18, 63
Maskenverarbeitung	31, 77, 80
Matrix (Speichermatrix, Bitmatrix)	4
maximaler (minimaler) aufspannender Baum	113
Maximumsuche (konventionell)	154
Maximum/Minimum (Vertikalverarbeitung)	58, 81
Mehrprozessoranlage (Mehrrechneranlage)	1, 81
Mikroassembler	11
Mikrobefehls-Vorrat	9
Mikroprogramm-Lader	12
Mikroprogramm-Testsystem	11
minimale Eliminationsordnung	124, 131
Minimum (Vertikalverarbeitung)	58
Modul-Konzept der INTERDATA 85	11, 82, 83, 88
Modul-Operationen ("Quersummen-Modul")	89
Modul-Operationen ("Vertikal-Modul")	85
Multiple-path-Verfahren	139
Multiplikation	76
Multiplikation sehr dünner Matrizen	101
Multiplikation (konventionell)	152
Multiplikation (Pipeline-Array)	25
Multiplikation (Vertikalverarbeitung)	45
Muster-Verarbeitung	69
nicht-lineare Operationen	29
Notation der Algorithmen	31, 107
OMEN	7, 17
Operationencodierung durch Masken	41, 44, 58, 83
Orthogonalspeicher	3, 16, 37, 69, 87, 111, 119, 149

Packen	14, 71, 86, 142, 149
Parallelität	30, 76, 79
Parallel-Arithmetik	4
perfekte Eliminationsordnung	124, 128
perfekter Eliminationsgraph	124, 128
Pipeline-Array	17, 23
Potenzenberechnung	100, 158
Pseudoparallelität	31
Push-down-stack	8
Quadratwurzel	52, 77
Quadratwurzel (Pipeline-Array)	25
Quadratwurzel (Vertikalverarbeitung)	52
Quadrieren (Pipeline-Array)	25
"Quersummen-Modul"	37, 56, 61, 67, 86
Relevanzmaske	63, 78
schnelle Fourier-Transformation	37, 99
Schwellwertvergleich (Vertikalverarbeitung)	62, 69
Selektionsmaske	38, 78
Selektorbits des assoziativen Speichers	18
Speicherbedarf	32
Speicherooperationen	10
Speicherprozessor	91
Speicherprozessor mit Spezialregistern	94
Speicherprozessor mit Warteschlangen	97
Speicherzugriffe	79, 80, 91
Sprachelemente	33
Sprachsymbole	35
STARAN	6, 17
starker Zusammenhang in einem Graphen	104, 105, 110
Trefferanzeige des assoziativen Speichers	18
Überlaufmarkierung	40
unkonventionelle Rechner und Rechnersysteme	1
Unterbrechbarkeit von Mikroprogrammen	10
Unterbrechungs-Konzept	11, 12
Unterbrechungs-System	11, 12
Vektoren-Addition (assoziativer Speicher)	23
Vektoren-Arithmetik	37
Vektor-Skalar-Verknüpfung	37
Vektor-Vektor-Verknüpfung	37
Verknüpfungsrichtung	30
vertikal codierte Mikrooperationen	9
"Vertikal-Modul"	43, 45, 65, 82
Warshall's Algorithmus	119
Wege in einem Graphen	119
Wegematrix	119
Zahlendarstellung	31
Zahlen addierend/subtrahierend (Vertikalverarbeitung)	62
Zeichendarstellung	32
zelluläres Array	102, 103, 113, 116, 123, 124, 144, 148
Zweierkomplement (Vertikalverarbeitung)	44

Lebenslauf

Geboren am 28. Mai 1941 in Rosenheim.

Volksschulbesuch in Rosenheim, Bücken (Kreis Grafschaft Hoya) und Freiburg, Gymnasium in Freiburg. 1960 Abitur.

Studium der Mathematik mit Physik als Nebenfach in Freiburg. 1967 Diplom.

Anschließend wissenschaftliche Mitarbeiterin an den Universitätsrechenzentren in Mainz und Ulm. Seit 1972 am Institut für Mathematische Maschinen und Datenverarbeitung der Universität Erlangen-Nürnberg.

Bisher sind folgende Arbeitsberichte des Instituts für
Mathematische Maschinen und Datenverarbeitung erschienen :
=====

Band 1
=====

- NR. 1 Schmitt, A. :
Theorie der nichtdeterministischen und unvoll-
ständigen Mealy-Automaten.
(Januar 1967)
- NR. 2 Göring, W., Henning, W., Rasche, W. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 1: Übersicht.
(Juli 1967)
- NR. 3 Namneck, P. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 2: Über Verfahren zur syntaktischen Analyse von
Sätzen formaler Sprachen.
(Juli 1967)
- NR. 4 Eggers, B. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 3: Möglichkeiten der automatischen Erkennung
und selbständigen Korrektur von syntaktischen Fehlern
in formalen Sprachen.
(Januar 1968)
- NR. 5 Schneider, H. J. :
Programmierung numerischer Probleme.
(Oktober 1967)
- NR. 6 Wolf, F. :
Untersuchungen über den Vorhersagefehler bei
stochastischen Automaten.
(Februar 1968)
- NR. 7 Klar, R. :
Rechenautomaten 1.
(April 1968)
- NR. 8 Schmitt, A., Fehling, P. :
Rechenautomaten 2.
(November 1968)
- NR. 9 Weiss, S. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 4: Ein Programmsystem zum Testen von Mustererken-
nungsverfahren unter Verwendung eines Sichtgerätes.
(Juli 1968)
- NR. 10 Dokumentation über die Arbeit des Instituts in den
Jahren 1966 bis 1968.
(Januar 1969)

Band 2

=====

- NR. 1 Vollmar, R. :
Über einen Automaten mit Speicherband.
(Januar 1969)
- NR. 2 Händler, W. :
Design Philosophy for Computer Systems.
(März 1969)
- NR. 3 Brackmann, U., Händler, W., Lange, M., Rasche, W.
Wolf, H. P. :
Beiträge zur Kommunikation Mensch - Maschine.
(April 1969)
- NR. 4 Haferkorn, J., Schmitt, A., Seidel, R., Wolf, F. :
Rechnergestützter Unterricht.
(April 1969)
- NR. 5 Böhme, J. F., Göring, W. :
Zwei Beiträge zur Mustererkennung.
(August 1969)
- NR. 6 Köhl, J. :
Datenreduktion zur Speicherung ebener Figuren.
(April 1969)
- NR. 7 Haferkorn, J., Henning, W., Rasche, W. :
Ein bildschirmorientiertes Betriebssystem für die
PDP 7.
(Mai 1969)
- NR. 8 Saffert, J. :
Mikroprogramm-Steuerwerke.
(Juni 1969)
- NR. 9 Spies, P. P. :
Rechnerstrukturen und Ablaufgeschehen in Rechen-
automaten - Analysis und Simulation.
(September 1969)
- NR. 10 Händler, W., Wolf, F. :
Bericht über eine USA-Reise - Aktuelles aus dem
Gebiet der Informatik.
(September 1969)
- NR. 11 Dokumentation über die Arbeit des Instituts im
Jahre 1969.
(Januar 1970)

Band 3

=====

- NR. 1 Schmitt, A. :
Bildschirme und Kommunikation Mensch - Maschine.
(Februar 1970)
- NR. 2 Böhme, J. F. :
Experimente mit stochastischen Automaten.
(Juli 1970)
- NR. 3 Schneider, H. J. :
Chomsky-Systeme für partielle Ordnungen.
(August 1970)
- NR. 4 Haas, G. :
Programme zur dialogischen Logik.
(Oktober 1970)
- NR. 5 Weicker, R. :
Erkennung von kontextfreien Sprachen durch
Tabulator-Turingmaschinen.
(Juli 1970)
- NR. 6 Schmitt, A. :
Optimale Vorhersage der Ausgabe stochastischer
Automaten über lange Zeiten.
(Februar 1971)
- NR. 7 Dokumentation über die Arbeit des Instituts im
Jahre 1970.
(Februar 1971)

Band 4

=====

- NR. 1 König, R. :
Reduktion und Zerlegung von Automaten.
(Juni 1971)
- NR. 2 Klar, R. :
Messung von Rechneraktivitäten.
(Juli 1971)
- NR. 3 Augustin, U. :
A Note on Arbitrary Varying Channels with
Noiseless Feedback.
A Note on Channels with Finite Memory.
(August 1971)
- NR. 4 Brüggmann, H. W. :
SIDIPRO, Dialogsprache und Echtzeit-Übersetzer
für die Programmierung eines Hochleistungs-
Sichtgerätes.
(August 1971)
- NR. 5 Händler, W. :
Strukturen von Computer-Graphics Systemen.
(Oktober 1971)
- NR. 6 Henning, W. :
Programmierung eines graphischen Sichtgerätes.
(April 1972)
- NR. 7 Dokumentation über die Arbeit des Instituts im
Jahre 1971.
(April 1972)

Band 5

=====

- NR. 1 Steierer, H. F. :
Algorithmen zur Zustandszerlegung von Automaten
unter besonderer Berücksichtigung der graphen-
theoretischen Aspekte.
(März 1972)
- NR. 2 Händler, W., Weizenbaum, J. :
Display Use for Man-Machine Dialog.
(September 1972)
- NR. 3 Dürre, K. :
Erfahrungen mit dem Programmiersystem LEKTOR-1
für Programmierten Unterricht.
(September 1972)
- NR. 4 Vollmar, R. :
SICELA - ein Interpretierer zur Simulation von
Automatennetzen.
(September 1972)
- NR. 5 Spies, P. P. :
Informatik III und Informatik IV.
Vorlesungsskriptum
(Oktober 1972)
- NR. 6 Görz, G. :
Die Sprache GEDANKEN als Beitrag zur syntaktischen
und semantischen Definition von Programmiersprachen.
(Oktober 1972)
- NR. 7 Buchzik, H. :
Maschinen-erzeugte Zufallszahlen.
(Dezember 1972)
- NR. 8 Dokumentation Über die Arbeit des Instituts im
Jahre 1972.
(April 1973)

Band 6

=====

- NR. 1 Domesle, R. :
Vergleich von top-down- und bottom-up-
Kellerautomaten.
(Dezember 1972)
- NR. 2 Brüggmann, H. W. :
Zugriffsstrategien bei Rotationsspeichern.
(Juli 1973)
- NR. 3 Fachgespräch über mehrdimensionale Sprachen.
Erlangen, 14.-15. Juni 1973
- NR. 4 Händler, W. :
Bericht über einen USA-Aufenthalt 1972/1973
Aktuelles aus dem Gebiet der Informatik.
The Concept of Macropipelining.
(Mai 1973)
- NR. 5 Müller, H. :
Diskrete algebraische Strukturen.
(September 1973)
- NR. 6 Müller, H. :
Programmschemata mit Procedures.
(Oktober 1973)
- NR. 7 Leeb, K. :
Pascal Theorie.
(Dezember 1973)
- NR. 8 Chatterji, B. N. :
Two-stage Reinforcement Learning System.
An Algorithm for Economical Storage in Dynamic
Programming Approach to Pattern Recognition.
(Dezember 1973)
- NR. 9 Dokumentation über die Arbeit des Instituts im
Jahre 1973.
(Januar 1974)

Band 7

=====

- NR. 1 Eichmeier, P., Goeb, H. :
Anwendungsbeispiele von H-Graphen und VDL
zur semantischen Beschreibung.
(April 1974)
- NR. 2 Händler, W. :
Unconventional Computation by Conventional
Equipment.
(Mai 1974)
- NR. 3 Keramidis, S. :
Ein Beitrag zur Theorie der Prozesssysteme.
(August 1974).
- NR. 4 Nagl, M. :
Formale Sprachen von markierten Graphen.
(August 1974).
- NR. 5 Göttler, H. :
ALGOL 68 - Eine exemplarische Einführung.
(November 1974)
- NR. 6 Altmann, W. :
Konstruktionen zur Erweiterbarkeit von Sprachen.
(November 1974)
- NR. 7 Bunke, H. :
Beschreibung eines syntaxgesteuerten inkrementellen
Compilers durch Graph-Grammatiken.
(Dezember 1974)
- NR. 8 Jahresbericht 1974 der Informatik.
(März 1975)

Band 8

=====

- NR. 1 Ehrig, H., Nagl, M., Schneider, H. J. :
Three Papers on Graph Rewriting Systems and their
Applications.
(Januar 1975)
- NR. 2 Müller, H. :
Diskrete algebraische Strukturen II.
(April 1975).
- NR. 3 Klar, R., Wichmann, H. :
Mikroprogrammierung.
(Juli 1975)
- NR. 4 Weicker, R. :
Turingmaschinen mit assoziativem Speicherzugriff.
(Juli 1975)
- NR. 5 Schneider, H. J. :
Lexikon zur Programmierung I (ALGOL 60).
(Juli 1975)
- NR. 6 Weitok, R. :
Ein neuer Interpreter-Algorithmus für LISP
und seine Korrektheit.
(Juli 1975)
- NR. 7 Luft, A. :
Grundlagen einer Methode für die rechnerge-
stützte Verifikation von while-Programmen
während der Programmerstellung.
(August 1975)
- NR. 8 Rohrer, H. :
A Supervised Network of Adaptive Automata for
Pattern Recognition.
Drewes, B. :
Implementation of a Structural Pattern Recognition
System.
(Dezember 1975)
- NR. 9 Klar, R., Schreiber, H. :
Messungen mit dem ZÄHLMONITOR II.
(Dezember 1975)
- NR. 10 Jahresbericht 1975 der Informatik.
(März 1976)

Band 9

=====

- NR. 1 Brendel, W., Frühauf, T., Nagl, M. :
Graph Rewriting Systems II.
Three Papers on Theory, Application, and Implementation.
(Februar 1976)
- NR. 2 Brüggmann, H. W., Rudert, A., Lehr, H. :
EBIS
Strukturierte Bildprogrammierung auf höherer Ebene.
(Februar 1976)
- NR. 3 Hofmann, W., Wichmann, H. :
Konzept der Vertikalverarbeitung.
Implementierung und Analyse vertikaler Algorithmen
auf einem mikroprogrammierten Rechner.
(Mai 1976)
- NR. 4 Jordan, H. F., Smith, B. J. :
The Hardware-Software Interface.
(Juni 1976)
- NR. 5 Gernoth, B., Schmidt, B. :
Das Modell MEM-SIM.
Beschreibung eines Simulationsmodells zur Unter-
suchung einer Rechanlage mit einer symmetrischen
Mehrprozessorkonfiguration und privaten Speichern.
(Oktober 1976)
- NR. 6 Grötsch, E. :
Vergleichende Studie über parallele Ersetzung
markierter Graphen.
Jahn, B. :
Eine Studie zur Implementierung von Graph-Grammatiken
unter Verwendung von Assoziativverarbeitung.
(Dezember 1976)
- NR. 7 Mertens, P., Neuwirth, W., Purucker., G.,
Schaller., K., Schmitt., W. :
Verknüpfung von Daten- und Methodenbanken, dargestellt
am Beispiel der Analyse von Marktforschungsdaten.
(Dezember 1976)
- NR. 8 Parallelismus in der Informatik.
Wissenschaftliches Kolloquium aus Anlass des zehnjährigen
Bestehens des IMMD, Erlangen, 10./11. Juni 1976.
(Dezember 1976)
- NR. 9 Jahresbericht 1976 der Informatik.
(März 1977)

Band 10

=====

- NR. 1 Hoffmann, W. :
Modelle und Analysen von Teilnehmerrechensystemen,
angewendet auf das Teilnehmersystems RESPOND unter
CD 3300 MASTER.
(Februar 1977)
- NR. 2 Körber, K., Werzinger, G. :
Das Modell SIM-LAB.
Simulation eines klinisch-chemischen Laboratoriums.
(März 1977)
- NR. 3 Altmann, W., Weber, D. :
Programmierungsmethodik.
(März 1977)
- NR. 4 Maehle, E. :
Fehlertolerante Rechnerstrukturen.
(April 1977)
- NR. 5 Beth, T. :
Kombinatorik: Theorie der Vertretersysteme.
(April 1977)
- NR. 6 Mertens, P. :
Einige Hinweise zur technischen Gestaltung von
Bildschirmdialogen.
(Mai 1977)
- NR. 7 Jäpel, D. :
Ein Begriffssystem zur Formulierung von Schutzaspekten
und seine Anwendung auf Programmsysteme.
(Juni 1977)
- NR. 8 Brendel, W., Grötsch, E., Nagl, M., Schneider, H.-J. :
Graphersetzungssysteme, Beiträge zur Theorie, Anwen-
dung und Implementierung.
(Juni 1977)
- NR. 9 Rüdel, W. :
Darstellung und Vergleich abstrakter Modelle zur
Beschreibung geschützter Systeme.
(Juli 1977)
- NR. 10 Bolch, G., Gall, R., Grassl, M., Schmidt, B. :
Modelle für Trommelspeicher.
(Juli 1977)
- NR. 11 v. Eichhorn, K. :
Steiner Systeme, Codes und affine Räume.
(Juli 1977)
- NR. 12 Göttler, H. :
Zweistufige Graphmanipulationssysteme für die
Semantik von Programmiersprachen.
(Juli 1977)

Band 10 (Forts.)

=====

- NR. 13 Keramidis, S. :
 Ein allgemeines formales Modell für Betriebs-
 systeme und dessen Implementierung.
 (September 1977)
- NR. 14 Arnold, F., Benker, L., Grütz, M. :
 Die Fast Fourier Transformation und Möglich-
 keiten ihrer Implementierung auf EGPA.
 (November 1977)
- NR. 15 Gardill, R., Händler, W., Hessling, H.,
 Klar, R., Spies, P. P. :
 ERES
 Eine nichtprozedurale Rechnerentwurfssprache
 mit präziser Zeitbeschreibung.
 (November 1977)
-

Offsetdruck-Fotodruck

J. Hogl, 8520 Erlangen, Hauptstr. 109 (Nähe Martin-Luther-Platz) Tel. 22990
